Operating System/2™

Version 1.3

Procedures Language 2/REXX User's Guide

Programming Family

Operating System/2™

Version 1.3

Procedures Language 2/REXX User's Guide

Programming Family

First Edition (September 1990)

Special Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

Operating System/2	OS/2
Systems Application Architecture	SAA

About This Guide

The *IBM Operating System/2 Procedures Language 2/REXX User's Guide* (referred to as *User's Guide* in the rest of this book) describes the programming language known as REXX. REXX is an integral part of IBM Operating System/2* Standard Edition, Version 1.3, and IBM Operating System/2 Extended Edition, Version 1.3. (Both programs are hereafter referred to as the OS/2* program.) This guide describes how to write programs using REXX.

Before You Begin

Before reading this book, it is important that you consider the following items:

What You Need

This is what you need to get started:

- Your computer with the OS/2 program installed.
- A text-editing or word-processing program such as the system editor supplied with the OS/2 program. Whatever editor you use must be able to create *straight-ASCII* files (nearly all such programs can).
- *IBM Operating System/2 Version 1.3 Procedures Language 2/REXX Reference.*

It is also useful, though not essential, to have a printer so you can print your programs.

What You Need to Know

To use this book, you should know:

- What a file is and how to create a text file
- How to use an editor or word processor
- The basic OS/2 commands for manipulating files, such as COPY, DELETE, DIR, and so on.

If you have little or no experience with the OS/2 program or with personal computers, you may want to read *IBM Operating System/2 Standard Edition, Version 1.3 Getting Started* or *IBM Operating System/2 Extended Edition, Version 1.3 Getting Started.* You can also find an introduction to REXX in *IBM Operating System/2 Standard Edition, Version 1.3 Using Advanced Features* or *IBM Operating System/2 Extended Edition, Version 1.3 User's Guide, Volume 1: Base Operating System.*

Who Should Read This Guide

Both inexperienced and experienced computer users should read this guide to learn about REXX. The guide shows how REXX is a useful programming language for both the experienced programmer and the user new to programming.

How This Guide Is Structured

Each chapter, except chapter 1, is divided into two parts:

- The first part of each chapter, *Basics*, is a tutorial guide to the most frequently used features of REXX. "Basics" in each chapter builds on the preceding chapters and prepares you for those that follow.
- The second part of each chapter, *Advanced Topics*, discusses more advanced information about REXX and includes descriptions and examples of the more specialized features of REXX.

Each chapter of this Guide concentrates on a single topic.

- Chapter 1, "Introduction to REXX," is a brief introduction to programming and REXX.
- Chapter 2, "How REXX Works," describes how the program works.
- Chapter 3, "Variables," tells about handling data in terms of symbols.
- Chapter 4, "Expressions," shows how REXX computes information.
- Chapter 5, "Commands," describes using REXX with OS/2 and other programs.
- Chapter 6, "Program Control," tells about refining and automating programs.
- Chapter 7, "Program Structure," tells about running programs within programs.
- Chapter 8, "Parsing," tells about reading and analyzing data.
- Chapter 9, "Arithmetic," shows how REXX calculates numbers.
- Chapter 10, "Input and Output," describes using REXX with outside data.
- Chapter 11, "Program Style," describes how to plan and correct programs.

Related Publications

The following related publications are included in the OS/2 Standard Edition library:

IBM Operating System/2 Standard Edition, Version 1.3 Getting Started
IBM Operating System/2 Standard Edition, Version 1.3 License Information
IBM Operating System/2 Standard Edition, Version 1.3 Product Information
IBM Operating System/2 Standard Edition, Version 1.3 Using Advanced Features.

The following related publications are included in the OS/2 Extended Edition library:

IBM Operating System/2 Extended Edition, Version 1.3 Getting Started
IBM Operating System/2 Extended Edition, Version 1.3 License Information
IBM Operating System/2 Extended Edition, Version 1.3 User's Guide, Volume 1: Base Operating System
IBM Operating System/2 Extended Edition, Version 1.3 Keyboard Layouts
IBM Operating System/2 Extended Edition, Version 1.3 Procedures Language 2/REXX Reference.

Contents

Figures

Chapter 1. Introduction to REXX

The REstructured eXtended eXecutor language, or REXX, is a versatile, easy to use structured programming language that is an integral part of the OS/2 program. Its simplicity makes it a good first language for beginners. For more experienced users and computer professionals, REXX offers powerful functions, extensive mathematical capabilities, and the ability to issue commands to multiple environments.

Features of REXX

The following REXX features contribute to its versatility and function.

Ease of Use

REXX is easy to learn and use because many instructions are meaningful English words. Unlike some programming languages that use abbreviations, REXX instructions are common words such as SAY, PULL, IF...THEN...ELSE, DO...END, and EXIT.

Free Format

REXX has few rules about format. A single instruction can span many lines or multiple instructions can be entered on a single line. Instructions do not have to begin in a particular column and can be typed in uppercase, lowercase, or mixed case. Spaces can be skipped in a line or entire lines can be skipped. There is no line numbering.

Interpreted

REXX is an interpreted language. When a REXX program runs, its language processor reads each statement from the source file and runs it, one statement at a time. Languages that are not interpreted must be compiled into machine language (in separate files) before they can be run.

Built-in Functions

REXX has built-in functions that perform various processing, searching, and comparison operations for both text and numbers. Other built-in functions provide formatting capabilities and arithmetic calculations.

Typeless Variables

REXX regards all data as character strings. This means that there is no need to predefine variables or variable arrays as strings or as numbers. REXX performs arithmetic operations on any string that represents a valid number, including those in exponential formats.

Parsing Capabilities

REXX includes capabilities for manipulating character strings. This allows programs to read and separate characters, numbers, and mixed input.

Debugging

REXX displays messages with meaningful explanations when a REXX program encounters an error. In addition, the TRACE instruction provides a powerful debugging tool.

REXX and the OS/2 Program

The most vital role REXX plays is as a procedural language for the OS/2 program. A REXX program can serve as a script for the OS/2 program to follow. By using REXX, long, complex, or repetitious tasks can be reduced to a single command or program that can be run from Presentation Manager.

REXX is a built-in feature of the OS/2 program, so programs are run directly from the Desktop Manager, File Manager, or OS/2 Windowed or Full Screen command prompt. There is no installation process or separate environment. Anywhere that an OS/2 command or batch file can be used, a REXX program can be run. Any REXX program can call OS/2 commands.

About REXX and SAA

REXX is one of the programming languages included in the IBM Systems Application Architecture* (SAA*). SAA is a framework of standards and definitions intended to promote consistency among different IBM products. Programs written in REXX according to SAA specifications are portable to all other SAA environments. For example, a REXX program written for OS/2 environment can be run in a CMS or TSO/E environment, if the program does not use OS/2-specific features.

To learn more about using REXX in SAA environments, see the *Common Programming Interface Procedures Language Reference.*

About Programming

A program is a list of instructions that have a basic sequence. Some instructions indicate actions and some instructions specify the number and sequence of these actions. There are also instructions to tell you how to execute other instructions. Those that specify repetitious actions are iterative. Those that indicate when an action should begin or end are conditional.

You may think of programming as a skill practiced only by computer experts, but that is not true. You do not need to know how a computer works to write a program. Anyone can write a program, and almost everyone who uses a computer eventually needs to do so. With a little programming knowledge, you can reduce a long or repetitious series of commands into a single command. You can also customize OS/2 programs and other programs to work in a way that will best suit your needs. Programming helps you make the computer work faster or better. That is what REXX was meant for, and this book should make REXX easier to understand.

If You Have Never Written a Computer Program

If you are inexperienced at programming, you will find it fairly easy to learn and write programs in REXX. Start by reading "Basics" in each chapter. After you have read "Basics," go back and read "Advanced Topics" in each chapter to learn more about specific topics.

If You Are Already Familiar With Another Language

If you are an experienced programmer, reading "Basics" in each chapter gives you an overview of the REXX language. Or, you may prefer to read about individual topics, one at a time. Here are some areas you may want to investigate:

- If you now use OS/2 CMD files to automate your work, you will find that REXX gives you more flexibility in controlling program flow and in handling parameters. Refer to Chapter 5, "Commands," which discusses how REXX works with OS/2 commands.
- If you are skilled in BASIC, you may want to note these ways that REXX differs from BASIC:
  - There is no line numbering.
  - There are no GOSUB or GOTO statements. Use CALL and SIGNAL instead (see "Subroutines" on page 7-1).
  - REXX variables have no data type.
  - There are no DIM arrays. Use compound variables instead (see "Using Compound Symbols" on page 3-8).
- If you are familiar with development languages, such as C and Pascal, you will find REXX somewhat similar. Again, the main difference is that a REXX program is interpreted; that is, the source code of the program is processed line by line. There is no compiling process.

  Refer to Chapter 5, "Commands," for examples of command-passing and Chapter 10, "Input and Output," for examples of file and queue processing.

Exercises and Examples

As with other programming languages, you do not learn REXX by reading about it. Exercises and examples are included to help you learn REXX by using it. To get the most out of this book:

- Test yourself with the exercises as you read.
- Examine the examples and sample programs in the text. Type them exactly as they are shown.
- Try your own variations of each program. See if you can find a different or better way to do what the sample program does.

Note: When your REXX program issues an OS/2 command, REXX passes the command to the OS/2 command handler for processing. This processing includes displaying the command on the screen (*echoing*).

For the sake of simplification, the examples in this book do not include echoing of the commands.

The REXX Reference

The *REXX Reference* contains a more complete description of how to use the REXX language. You should have a copy of this book, so you can look up any instruction or function not completely defined here.

Think of the *REXX Reference* as a dictionary for REXX and this *User's Guide* as a book of directions and ideas.

Chapter 2. How REXX Works

A REXX program is a list of instructions for your computer. The program is simply a text file that you create with a text-editing or word-processing program. Sometimes a computer runs a program with no guidance. Other times it may need additional information from the user to do its work. One way that a computer communicates with the user is to ask questions and then compute results based on the responses. The programmer (you) can include instructions that let the computer converse with the user.

Basics

In this chapter:

Basics

- A computer conversation—creating and running your first REXX program and how REXX interprets it
- What goes into a program—elements in the grammar of REXX.
- Syntax errors—how to read REXX messages.

A Computer Conversation

Figure 2-1 shows a sample REXX program. It asks users to type their names. Then the program greets the user by that name. For example, if the user types Jean, the program displays `Hello JEAN` on the screen. If the user does not type anything, the program displays `Hello stranger!` on the screen.

```
/* A conversation */
say "Hello! What is your name?"
pull who
if who = "" then say "Hello stranger!"
else say "Hello" who
```

Figure 2-1. HELLO.CMD

This sample program consists of five statements called *clauses*. The various pieces of this program are:

`/* ... */`	The first clause is a *comment* explaining what the program is about. All REXX programs must start with a comment beginning in column 1. Except for this, all other comments are ignored.
`say`	The second clause is the *keyword instruction* `say`, which displays text on the screen.
`"Hello!..."`	Anything in quotes after `say` is displayed on the screen exactly as it was typed. This is called a *literal string*.
`pull`	The third clause is the keyword instruction `pull`, which reads and stores the response typed by the user of the program.

`who`	This is a *variable*: a name given to the place in memory where the user's response is stored.
`if`	The fourth clause is the keyword instruction `if`, which tests for a given condition.
`who = ""`	The condition to be tested: whether the variable `who` is empty.
`then`	Tells the program to process the instruction that follows, if the tested condition *is* true.
`say "Hello stranger!"`	Displays `Hello stranger!` on the screen if the condition is true.
`else`	This final clause gives an alternative direction: process the instruction that follows, if the tested condition *is not* true.
`say "Hello" who`	Displays `Hello`, followed by whatever data is stored in `who`, if the tested condition is not true.

Creating Your First Program

Follow these steps to create your first program by:

1. Using a word processor or editor to create a text file named HELLO.CMD. Be sure to make it a *straight-ASCII* or *nondocument* file, without special formatting characters.
2. Typing the HELLO.CMD program exactly as shown in Figure 2-1 on page 2-1. Be sure that the first line begins with /* and ends with */.
3. Saving the file and then returning to Presentation Manager.

Running the Program

Follow these steps to run the sample program from the OS/2 command prompt by:

1. Typing the file name of the program. In this example, type `hello` at the OS/2 prompt and press the Enter key.
2. Typing your name and pressing the Enter key. If your name is Fred, `Hello FRED` is displayed on the screen.

```
[C:\] hello
Hello! What is your name?
?
fred
Hello FRED
[C:\]
```

When you run this program:

1. The SAY instruction displays `Hello! What is your name?` on the screen.
2. The question mark means that the PULL instruction has stopped the program, waiting for your response.
3. You type `fred` on the command line and press the Enter key.
4. The PULL instruction puts `FRED` into the variable (the place in memory) called `who`.

5. The IF instruction tests, is who equal to nothing?

   ```
   who = ""
   ```

   To find out, REXX substitutes the stored value for the variable name. Now the question is, is FRED equal to nothing?

   ```
   "FRED" = ""
   ```

6. Not true. The SAY instruction after then is not processed. Instead, REXX processes the SAY instruction after else.
7. The SAY instruction displays "Hello" who, which is evaluated as Hello FRED on the screen.

The following is displayed on the screen, if you press the Enter key without typing a response.

```
[C:\] hello
Hello! What is your name?
?

Hello stranger
[C:\]
```

When you run this program:

1. The PULL instruction puts nothing ("") into the variable (the place in memory) called who.
2. The IF instruction tests is who equal to nothing?

   ```
   who = ""
   ```

   When the stored value of who is substituted, this is:

   ```
   "" = ""
   ```

3. This time, it is true. The SAY instruction after then is processed, and the SAY instruction after else is not.

Problems

Did you get your version of HELLO.CMD to run? If not, check that you have correctly typed it in. Also, be sure that you used the nondocument mode of your text editor to create the program file.

The most common error is forgetting the comment on the first line. Be sure that the first line begins with the /* and ends with the */ characters. If you mistype or omit these characters, you get a message on the screen from REXX that looks something like this.

```
[C:\] hello
     0 +++;
REX0006: Error 6 running C:\HELLO.CMD, line 0: Unmatched "/*" or quote
```

This means that REXX found the beginning /* but not the ending */ of the comment. Edit your HELLO.CMD program file to match the sample in Figure 2-1 on page 2-1.

If you do not start the first line with /*, you get a message like this.

```
SYS1041: The name specified is not recognized as an
internal or external command, operable program or batch file.
```

This means that the operating system did not recognize HELLO.CMD as a REXX program. (See "First-line Comments" in the following text.)

If you get another error, refer to "Fixing Syntax Errors" on page 2-6.

Stopping a Program

If you need to stop a program, press the Control (Ctrl)+Break keys. That is, press and hold down the Ctrl key and then press the Break key once. REXX stops running the program and returns to the OS/2 command prompt.

What Goes into a Program

To explain what happens when you run a REXX program, a number of terms have been introduced. There will be more; so before continuing, here are definitions of the terms used so far.

Comments

When you write a program, you will want to read it later (for example, before improving it). Other users of your program will also want to read it to know what the program is for, what kind of input it can handle, what kind of output it produces, and so forth. You may also want to write remarks about individual instructions. All these things, words that are to be read by people but not interpreted by REXX, are called *comments*.

To indicate comments, use /* to mark the start of a comment and */ to mark the end of a comment.

The /* causes REXX to stop interpreting the program. Interpreting starts again only after a */ is found, which may be a few words or several lines later. For example:

```
/* This is a comment. */

say ...  /* This is a comment on the same line as the instruction */

/* Comments may
   occupy more
   than one line. */
```

First-line Comments: The first line of a REXX program *must* start with a comment. The OS/2 program can be programmed with its built-in Batch Facility and in REXX. Both Batch Facility and REXX programs can use the file name extension CMD. Each type requires its own special processing, so the OS/2 program checks the first line of the program. If it finds a REXX-style comment, the program is processed as REXX. Therefore, to recognize that your program is written in

REXX, the first line of the file must be or begin a comment. For example:

```
/* this is a REXX program.  */
```

Also, the first-line comment must begin in column 1. It is sufficient to use /* */, but a better use for the space is to give a brief description of your program. You can also do it this way.

```
/************************************
*  HELLO.CMD written by J. Smith     *
*       June 30, 1989                *
* A program to greet a user by name. *
************************************/
```

Keyword Instructions

Words such as SAY, PULL, and IF are part of the REXX language called *instructions*. The words themselves are referred to as *keywords*. You will notice that they are usually verbs. They are the directions that tell REXX what to do with information at a certain point in the program.

SAY (display on screen) hello.
PULL (accept and store) information from the user.
IF (test) this situation is true, then perform this action.

When you list these instructions in the order you want REXX to execute them, you have created a program.

Clauses: A REXX program is made up of *clauses*; that is, complete instructions, including the information it works on and any options that may be used. REXX reads each clause and processes it before going on to the next. That is why REXX is an *interpreted* language.

In the previous sample program, each line of text corresponds to a single clause. REXX allows exceptions to this (see "More about Clauses" on page 2-10). The examples and sample programs in this book follow the convention of one clause to a line, except where noted.

Literal Strings

When REXX finds a quote (either " or '), it stops processing and looks ahead for the matching quote. The string of characters inside the matching quotes is used as it is and is called a *literal string*. Examples of literal strings are:

```
'Hello'
"Final result:"
```

If you need to use quotation marks within a literal string, use quotation marks of the other type to delimit the string. For example:

```
"Don't panic"
'He said, "Bother"'
```

There is another way. Within a literal string, a pair of quotes (the same type that delimits the string) is interpreted as one of that type. For example:

```
'Don''t panic'                    (same as "Don't panic")
"He said, ""Bother"""          (same as 'He said, "Bother"')
```

Uppercase Translation: When a clause is processed, any letters that are *not* in quotes are translated to uppercase. For example, the letters `a, b, c, ... z` are changed to `A, B, C, ... Z`.

REXX also ignores some of the spaces that you may have written into your program, keeping only one space between words. Figure 2-2 shows an example using quotes to get more than one space between words.

```
/* Example: cases and spaces */
say  Hello!     What     is     your     name?

say "Hello!     What     is     your     name?"

say Hello!"    "stranger!
```

Figure 2-2. HELLO2.CMD

The following is displayed on the screen, when you run the HELLO2 program.

```
[C:\] hello2
HELLO! WHAT IS YOUR NAME?
Hello!     What     is     your     name?"
HELLO!    STRANGER!
[C:\]
```

Note: In the HELLO.CMD sample program, the user's input `fred` was changed to `FRED`. That translation is not the process described here, but is a feature of the PULL instruction. The PULL instruction always converts the input to uppercase, which allows the user to type any combination of uppercase and lowercase letters.

Variables

When you need to work with changeable information (such as the user's name in HELLO.CMD), you can reserve a place in memory to store it. That place is called a *variable.*

When REXX processes a clause containing a variable, it substitutes the stored data for the variable. That is how the stored entry `FRED` took the place of the variable name `who` in the HELLO.CMD program.

Refer to Chapter 3, "Variables," for more information on variables.

Fixing Syntax Errors

The rules governing the arrangement of words and punctuation marks in a language are called *syntax.* The actions described are part of the syntax for the REXX language. If REXX encounters something that does not make sense according to its syntax, it stops running your program, displays the incorrect instruction line and an *error message* saying what is wrong, and returns to the OS/2 program.

Figure 2-3 shows the HELLO.CMD program with a syntax error. The */ is missing at the end of the second comment.

```
/* A conversation */
say "Hello! What is your name?"
pull who                                  /* Get the answer!
if who = "" then say "Hello stranger!"
else say "Hello" who
```

Figure 2-3. HELLO.CMD with a syntax error

The following is displayed on the screen, when you run the program and enter fred.

```
[C:\] hello
Hello! What is your name?
?
fred
     3 +++  Pull who;
REX0006: Error 6 running C:\HELLO.CMD,line 3: Unmatched "/*" or quote

[C:\]
```

This *error message* means:

- REX0006: is the REXX error number. If you need more information, type help followed by the error number (REX0006) at the command prompt and press the Enter key. More information about the error is displayed. You can also find error message help in the *REXX Reference*.
- The phrase in line 3 means REXX was processing the clause that started on line 3 when the error occurred.

Leaving out a final quotation mark at the end of a literal string causes REXX to issue a similar error message.

Test Yourself

1. Read the following program and write down what each clause is and what REXX will do with it, depending on how the user responds.

```
/* Who Am I? game */
say "What is my name?"
pull guess
if guess = "REXX" then say "You win!"
else say no but guess "is a good guess."
```

Figure 2-4. WHOAMI.CMD

   Create a file called WHOAMI.CMD, type the program shown in Figure 2-4, and run it.

   Did everything happen as you expected? If not, read this chapter again and then study the explanation below.

2. Figure 2-5 shows a program with an error in it. Create a file called TROUBLE.CMD, type the program, and run it.

```
/* Example: a syntax error */
say Unfortunately, there is an error here
```

Figure 2-5. TROUBLE.CMD

Using the error number, find the cause of the error in the *REXX Reference.* Correct the error and run the program again.

Answers:

1. The syntax of the WHOAMI.CMD program shown in Figure 2-4 on page 2-7 is:
   - `/* Who Am I? game */` is a *comment* describing the program. (The first line of a REXX program must start with a comment.)
   - `say` is an instruction that displays, `What is my name?`
   - `pull` is an instruction that stores the user response in the variable guess
   - `if` is an instruction that tests to see if the user typed `REXX`.

     Note: Because `pull` translates the entry to uppercase, you can type it in any combination of uppercase and lowercase letters (`rexx`, `Rexx`, `rExX`, and so on).
   - If `guess = REXX`, then `say` displays `You win!`.
   - If the user types something other than `REXX`, then the clause beginning with `else` is interpreted and `say` displays the result as follows:
     - `no` but is a string, but it is not in quotes. Therefore, it is changed to uppercase and is displayed as `NO BUT`.
     - `guess` is the name of a variable. The user's response, translated to uppercase, is substituted.
     - `"is a good guess."` is a literal string. It is displayed as is, even though `guess` is also the name of a variable.

   The following is displayed on the screen, if the user guesses correctly.

```
[C:\] whoami
What is my name?
?
rexx
You win!
```

The following is displayed on the screen, if the user guesses incorrectly.

```
[C:\] whoami
What is my name?
?
spot
NO BUT SPOT is a good guess.
```

The following is displayed on the screen, if the user presses the Enter key without typing a response.

```
[C:\] whoami
What is my name?
?

NO BUT  is a good guess.
```

The variable guess was empty, so the say instruction displayed nothing. Only the blank before and after the variable in the program remain.

That last response does not make much sense. See if you can think of a way to fix WHOAMI.CMD so that it does. (Hint: Take another look at HELLO.CMD.)

2. The error number for the program TROUBLE.CMD is 37. The error message displays `Unexpected "," or ")"`.

   REXX found a comma where it did not belong. It may not be obvious what to do about it. When you get a message like this, refer to the *REXX Reference* for a list of error messages and explanation of their causes.

   The comma has a special meaning when it is used outside of a literal string (see "More about Clauses" on page 2-10). Figure 2-6 shows that to use a comma as it is intended, it must be enclosed in matching quotes.

```
/* Example: a syntax error fixed */
say Unfortunately"," there is an error here
```

Figure 2-6. TROUBLE2.CMD

Summary

This completes "Basics" in this chapter. You have learned that REXX reads a literal string. In addition, you have learned how to:

- Write a program
- Run a program
- Use the SAY and PULL instructions.

"Advanced Topics" in this chapter discusses more about the structure of REXX programs.

To continue with "Basics," go to page 3-1.

Advanced Topics

In this chapter:

Advanced Topics

- Kinds of clauses—more about REXX syntax.

More about Clauses

Your REXX program is made up of a number of clauses. REXX processes clauses, one at a time, reading from left to right.

Usually, each clause occupies one line of the program, but that is only a convention. It is sometimes useful to be able to write more than one clause on a line or to extend a clause over many lines. The rules for writing clauses are:

- If you want to put more than one clause on a line, you must use a semicolon (;) to tell REXX where one clause ends and the next begins.
- If you want a clause to span more than one line, you must put a comma (,) at the end of the line to tell REXX that the clause continues on the next line. A comma used in the middle of a string is interpreted as part of the string itself. A comma inside a comment is ignored.

What is displayed on the screen, when the program shown in Figure 2-7 is run?

```
/* Example: there are six clauses in this program */
say "Everybody cheer!"
say "2"; say "4"; say "6"; say "8";
say "Who do we",
"appreciate?"
```

Figure 2-7. RAH.CMD

If you are not sure, create a file called RAH.CMD, type the program, and run it.

Types of Clauses

The three types of clauses are:

- *Null clauses*
- *Labels*
- *Instructions.*

Null Clauses

A clause that is empty (a blank line) or consists only of blanks or comments is called a *null clause*. REXX ignores all null clauses, except to check for a comment in the first line of a program. This means you can use spaces and blank lines to make your program more readable without affecting its performance.

Labels

Labels are symbols that mark positions or portions of a program, internal subroutines, condition traps, and so forth. They are distinguished by a trailing colon (for example, `ERROR:`). Except for their use with the CALL and SIGNAL instructions and for internal function calls, labels are regarded as null clauses.

Unlike null and instruction clauses, labels are self-delimiting. They do not require semicolons or *carriage returns* to separate them from other clauses.

Instructions

The three types of instruction clauses are:

Keywords Clauses that begin with words, such as PULL and SAY, that REXX recognizes as instructions. Keywords are not reserved words, but the language processor recognizes them by their context (see "Variables as Symbols" on page 3-7). Certain keyword instructions may comprise one or more clauses, such as the IF instruction.

The keyword instructions are listed alphabetically in the *REXX Reference*.

Assignments Clauses that assign values to variables. An assignment normally takes the form `symbol = expression`. The PARSE instruction and its variants PULL and ARG also assign values to variables. Refer to Chapter 3, "Variables," for a description of variable assignments.

Commands Clauses that are processed by other programs. A clause that is an expression by itself is interpreted as a command to be passed to the current environment (the application that initially called REXX). You can also use the ADDRESS instruction to pass commands to other environments. Refer to Chapter 5, "Commands," for a description of more commands.

For More Information

For a more complete discussion of REXX syntax, refer to "General Concepts" in the *REXX Reference*.

Chapter 3. Variables

Variables are a means of handling changeable information by representing it in terms of *symbols*. This chapter explains why variables are important when writing programs and describes the basic rules for using them.

Basics

In this chapter:

Basics

- Handling data with symbols
- Assignments
- Naming variables
- Other assignments.

Handling Data with Symbols

One basic requirement of any program is that it must work with information that is unknown when the program is written.

You could write a program that totals a fixed list of numbers. For example:

```
say "2 + 3 equals" 2 + 3
```

This example displays the result 5 each time you run it, but that is all you would get. This is a reliable program but not a very useful one. A program that is more useful can process different information each time it is run. You can do this by using variables to stand in for values to be processed. A variable is a symbol (one or more characters) that represents a value.

Figure 3-1 shows a program that contains a simple calculation.

```
/* the sum of two numbers */
say "Type a number:"
pull first                          /* waits for entry */
say "Type another number:"
pull second                         /* waits for entry */
say "The sum is" first + second
```

Figure 3-1. ADD2NUM.CMD

The following is displayed on the screen, when you run the program.

```
[C:\]add2num
Type a number:
?
25
Type another number:
?
32
The sum is 57
[C:\]
```

Two PULL instructions are used, allowing the user to type the two numbers to be added and then *assign* (store) them in the variables `first` and `second`. The SAY instruction displays the sum of the two.

Names and Values

The information stored in a variable is called its *value*. The value can be one or more words of text, numbers, or nothing. The value of a variable can change any time you want it to. It can be different each time the program is run, or it can change many times in a single run. If the value of a variable changes, the *name* of the variable stays the same. The variable names only have to be meaningful to you.

You can think of a variable as a name for the type of values you want it to hold.

Assignments

An instruction that stores a value in a variable or changes its value is called an *assignment*. The simplest form of assignment is the equal sign, a REXX clause in the form `name = value` where:

`name` is the name you give the variable

`value` is the value it will hold.

In more formal terms, the syntax of an assignment is in the form `symbol = expression` where:

`symbol` is a valid variable name

`expression` is the information to be stored, such as a number, a string, or some calculation that you want REXX to perform.

REXX *evaluates* (computes) the `expression` and then puts the result of that evaluation into the variable called `symbol`. The assignment instruction means "Evaluate the `expression` and store the result as `symbol`."

In an assignment, you name a variable and give it a value. For example:

- To give a variable called `total` the value `0`, use the assignment `total = 0`.
- To give another variable, called `price`, the same value as `total`, use the assignment `price = total`.
- To give the variable called `total` a new value, the old value of `total` plus the value of `something`, use the assignment `total = total + something`.

In a different type of assignment, `pull something`, the PULL instruction gives the variable `something` a value that the user types while the program is running.

Displaying the Value of a Variable

To display the value of a variable while a program is running, use the SAY instruction, as shown in Figure 3-2.

```
/* some assignments */
amount = 100                        /* assigns 100 to AMOUNT      */
money = "dollars"                   /* assigns "dollars" to MONEY */
say amount money                    /* displays "100 dollars"     */
amount = amount + 25                /* adds 25 to AMOUNT          */
say amount money                    /* displays "125 dollars"     */

/* Now get some input from the user */

say "Type a line, then press the Enter key" /* prompts the user to type  */
pull anything                      /* waits for user to press the Enter key*/
say "You typed:" anything           /* displays the input on screen  */
```

Figure 3-2. ASSIGN.CMD

If you use a SAY instruction with a variable that has not been assigned a value, some languages would generate an error. In REXX, the default value of a variable is its own name, converted to uppercase letters, as shown in Figure 3-3.

```
/* display unassigned variables */
say amount          /* displays AMOUNT */
say first           /* displays FIRST  */
say price           /* displays PRICE  */
say who             /* displays WHO    */
```

Figure 3-3. NOASSIGN.CMD

Note: Another way to display the value of a variable while a program is running is with the TRACE instruction, used for correcting programs. Refer to "Tracing Evaluation" on page 4-7.

Naming Variables

You can name variables almost anything you want. There are a few rules that REXX imposes and a few conventions that should be observed. A variable name can be any *symbol* (group of characters), containing up to 250 characters, with the following restrictions:

- The first character must be `A-Z`, `a-z`, `!`, `?`, or `_` . REXX translates lowercase letters to uppercase before using them.
- The rest of the characters may be `A-Z`, `a-z`, `!`, `?`, `_`, `.`, or `0-9`.
- The period (.) has a special meaning for REXX variables. Do not use it in a variable name until you understand the rules for forming compound symbols. See "Using Compound Symbols" on page 3-8.

Here are some tips for good programming practice:

- Give variables names that describe the data they represent.
- Give variables names that are different from REXX keywords or OS/2 commands.

- Give variables names that will not be confused with each other.
- Do not abbreviate unnecessarily. It is better that the name is long rather than obscure.
- Use each variable for only one purpose. Do not use the same variable for a user entry that you used elsewhere to accumulate a total.

Test Yourself

Which of the following could be used as the name of a REXX variable?

1. DOG
2. K9
3. 9T
4. nine_to_five
5. ?7

Answers:

1. OK
2. OK
3. Invalid, because the first character is a numeric digit.
4. OK, same as NINE_TO_FIVE
5. OK

Other Assignments

You can also use variables to store information that is unknown when you are writing the program.

Assigning User Input

One use for variables that has already been discussed is as a holding place for information typed by the user. The PULL and ARG keyword instructions are commonly used for this purpose.

The PULL instruction pauses the running of a program to let you type one or more items of data, which are then assigned to variables. For example, the PULL instruction was used in the program shown in Figure 3-1 on page 3-1 to get two numbers to add.

```
say "Type a number:"
pull first                  /* waits for entry */
say "Type another number:"
pull second                 /* waits for entry */
```

Each PULL instruction pauses the program and displays a ? to prompt you to type a number and press the Enter key. It then assigns the entry to the variable named in the instruction.

You can also use the PULL instruction to collect more than one item in an entry as long as the items are separated by spaces. The four lines in the preceding example could be replaced with:

```
say "Type two numbers (leave a space between) and press the Enter key"
pull first second
```

The PULL instruction pauses the program and displays a ? so you can type the two numbers to be added. When you press the Enter key, PULL reads the two numbers and assigns them, in the order they were typed, to the list of variables (`first` and second). This process of reading and separating information is called *parsing*.

The ARG instruction is another way to assign data from the user. ARG works similar to PULL, except that items are typed at the command prompt with the program name. The calculation in the program shown in Figure 3-1 on page 3-1 could also be done by the program shown in Figure 3-4, which follows.

```
/* the sum of two numbers, this time    */
/* typed at the command prompt          */
arg first second     /*collects entries */
say "The sum is" first + second
```

Figure 3-4. ADD.CMD

The following is displayed on the screen, when you run ADD.CMD.

```
[C:\]add 25 32
The sum is 57
```

Notice that with the ARG instruction there is no ? prompt because there is no PULL instruction to stop the program for the user to type a response. The numbers to be added are typed with the ADD command that starts the program.

Assigning an Expression Result

The instruction amount = amount + 25 in the program shown in Figure 3-2 on page 3-3 shows how variables can represent another type of unknown information—data that must be calculated or otherwise manipulated. You can assign to a variable the result of a calculation or *expression*, as shown in Figure 3-5.

```
/* area of a 3 by 5 in. rectangle */
area = 3 * 5
say area "sq. in."                  /* displays "15 sq. in." */

/* area of a 5 in. circle */
diameter = 5
radius = diameter/2
area = 3.14 * radius * radius
say area "sq. in."             /* displays "19.6250 sq. in." */
```

Figure 3-5. AREAS.CMD

REXX expressions can have very complex forms and they can work with all kinds of information.

Summary

This completes "Basics" in this chapter. You have learned how to:

- Assign a value to a variable using the equal sign
- Display the value of a variable
- Name variables
- Assign user input to a variable.

"Advanced Topics" in this chapter discusses:

- How REXX recognizes and processes variables
- Using variables in ordered groups called *arrays*
- Using variables in complex programs.

To continue with "Basics," go to page 4-1.

Advanced Topics

In this chapter:

Advanced Topics

- Variables as symbols
- Using compound symbols
- Variables in programs and subroutines
- Other types of data storage.

Variables as Symbols

Variables are part of a class of REXX language elements called *symbols*. These include:

- REXX keywords and instructions
- Labels used to call internal subroutines (see "CALL Instruction" on page 7-2)
- Constants
- Variables.

REXX uses the context of a symbol to determine if it is a keyword, a label, or a variable. For each symbol it encounters, REXX takes the following steps to determine how it will be handled:

1. If the first *token* is a symbol and is followed by:
   a. An equal sign (=), the clause is an assignment instruction. The symbol is a variable and is assigned the expression that follows the equal sign.
   b. A colon (:), the clause is a label, signalling the beginning of a subroutine.
2. If the symbol is in the list of REXX keyword instructions or is a keyword used in a control structure, such as while or then, REXX interprets the keyword accordingly. (See Chapter 6, "Program Control.")
3. If the symbol begins with a number, it is a constant (an unchangeable value).

If none of these steps determine how the symbol is to be handled, REXX evaluates the symbol as a variable and replaces the variable name with the stored value of the symbol.

Constants and Variables

Symbols that begin with a digit (0-9), a period, or a sign (+ or −) are *constants*. They cannot be assigned new values and, therefore, cannot be used as variables. Some examples of constants are:

77	a valid number
.0004	begins with a period (decimal point)
1.2e6	Scientific notation (equal to 1 200 000)
42nd	Not a valid number; its value is always 42ND.

All valid numbers are constants, but not all constants are valid numbers. The symbol `3girls` is not a valid number; it cannot be used as a variable name. Its value is always 3GIRLS.

The default value for a symbol is its own name, translated into uppercase letters. A variable that has not been assigned a value contains this default value.

Using Compound Symbols

There is a special class of symbols, called *compound symbols*, in which variables and constants are combined to create groups of variables for easy processing. A variable containing a period is treated as a compound symbol. Some examples of compound symbols are:

```
fred.3
row.column
array.I.J.
gift.day
```

You can use compound symbols to create a collection of variables that can be processed by their *derived names*. An example of a collection is:

```
gift.1 = 'A partridge in a pear tree'
gift.2 = 'Two turtle doves'
gift.3 = 'Three French hens'
gift.4 = 'Four calling birds'
:
```

If you know what day it is, you know what gift will be given. Assign a variable called DAY a value of 3.

```
day = 3
```

Then the instruction say gift.day, in the program shown in Figure 3-6 on page 3-9, displays Three French hens on the screen. When the program is run:

1. REXX recognizes the symbol gift.day as compound because it contains a period.
2. REXX checks if the characters following the period form the name of a variable. In this example, it is the variable name day.
3. The value of day is substituted for its name, producing a *derived name* of gift.3.
4. The value of the variable gift.3 is the literal string 'Three French hens'.

If day had never been given a value, its value would have been its own name, day and the derived name of the compound symbol gift.day would have been GIFT.DAY.

Figure 3-6 is a collection of consecutively numbered variables, sometimes called an *array*.

```
/* What my true love sent ... */

/* First, assign the gifts to the days   */
gift.1  = 'A partridge in a pear tree'
gift.2  = 'Two turtle doves'
gift.3  = 'Three French hens'
gift.4  = 'Four calling birds'
gift.5  = 'Five golden rings'
gift.6  = 'Six geese a-laying'
gift.7  = 'Seven swans a-swimming'
gift.8  = 'Eight maids a-milking'
gift.9  = 'Nine ladies dancing'
gift.10 = 'Ten lords a-leaping'
gift.11 = 'Eleven pipers piping'
gift.12 = 'Twelve drummers drumming'

/* list all gifts from the 12th day to   */
/* the 1st day; refer to the discussion  */
/* of loops on page 6-11.                */
do day=12 to 1
 say gift.day
end

/* now display the gift for a chosen day */
say "Type a number from 1 to 12."
pull day

/* check for proper input */
/* see page 9-1          */
if  \ datatype(day,n) then      /* if the entry is not a number */
  exit                          /* then exit the program        */

if day < 1 ¦ day > 12 then     /* same if it is out of range    */
 exit

say gift.day
```

Figure 3-6. TWELVDAY.CMD

Test Yourself

1. Write a program to display the days of the week repeatedly, as:

   ```
   Sunday
   Monday
   Tuesday
   Wednesday
   Thursday
   Friday
   Saturday
   Sunday
   Monday
   ⋮
   ```

   You will need to create an *endless loop*, using the DO instruction. (See "Repetitive Tasks" on page 6-11 for information about loops.)

   Note: To stop this program, press the Control (Ctrl)+Break keys. This stops any REXX program.

2. Extend this program to display the days of the month, as:

   ```
   Sunday 1st January
   Monday 2nd January
   ⋮
   ```

Answers:

1. Figure 3-7 shows one solution.

```
/* To display the days of the week indefinitely */
do forever
   say "Sunday"
   say "Monday"
   say "Tuesday"
   say "Wednesday"
   say "Thursday"
   say "Friday"
   say "Saturday"
end
```

Figure 3-7. DAYS1.CMD

In view of the preceding discussion, Figure 3-8 shows a solution that uses compound variables.

```
/* to display the days of the week indefinitely */
day.1 = "Sunday"
day.2 = "Monday"
day.3 = "Tuesday"
day.4 = "Wednesday"
day.5 = "Thursday"
day.6 = "Friday"
day.7 = "Saturday"

do j = 1
   say day.j
   if j = 7 then j = 0
end
```

Figure 3-8. DAYS2.CMD

2. Figure 3-9 shows how to extend the idea using the SELECT instruction.

```
/* to display the days of the month for January */
day.1 = "Sunday"
day.2 = "Monday"
day.3 = "Tuesday"
day.4 = "Wednesday"
day.5 = "Thursday"
day.6 = "Friday"
day.7 = "Saturday"

do dayofmonth = 1 to 31
   dayofweek = (dayofmonth+6)//7 + 1

   select
      when dayofmonth = 1 then th = "st"
      when dayofmonth = 2 then th = "nd"
      when dayofmonth = 3 then th = "rd"
      when dayofmonth = 21 then th = "st"
      when dayofmonth = 22 then th = "nd"
      when dayofmonth = 23 then th = "rd"
      when dayofmonth = 31 then th = "st"
      otherwise th = "th"
   end

   say day.dayofweek dayofmonth||th "January"
end
```

Figure 3-9. MONTH1.CMD

A Scoreboard Array

Figure 3-10 shows how you can use compound symbols to collect and process data. In the first part of the program, the first player's score is entered into SCORE.1, the second player's into SCORE.2, and so on. By using compound symbols, the SCORE array is processed to give the result in the required form.

```
/* This is a scoreboard for a game.  Any number of      */
/* players can play.  The rules for scoring are these:  */
/*                                                      */
/* Each player has one turn and can score any number of */
/* points;  fractions of a point are not allowed.  The  */
/* scores are entered into the computer and the program */
/* replies with                                         */
/*                                                      */
/*      the average score (to the nearest hundredth of  */
/*                         a point)                     */
/*      the highest score                               */
/*      the winner        (or, in the case of a tie,    */
/*                         the winners)                 */

/*------------------------------------------------------*/
/* Obtain scores from players                           */
/*------------------------------------------------------*/
say "Type the score for each player in turn.  When all"
say "have been typed, enter a blank line!"
say
n=1
do forever
  say "Please type the score for player "n
  pull score.n
  select
    when datatype(score.n,whole) then n=n+1
    when score.n="" then leave
    otherwise  say "The score must be a whole number."
  end
end

n = n - 1                  /* now n = number of players */
if n = 0 then exit
/*------------------------------------------------------*/
/* compute average score                                */
/*------------------------------------------------------*/
total = 0
do player = 1 to n
   total = total + score.player
end

say "Average score is",
    format(total/n,,2,0) /* format "total/n" with       */
                         /*   no leading blanks,        */
                         /*   round to 2 decimal places,*/
                         /*   do not use exponential    */
                         /*     notation                */

                                  /* continued ...   */
```

Figure 3-10 (Part 1 of 2). GAME.CMD

```
/*---------------------------------------------------*/
/* compute highest score                             */
/*---------------------------------------------------*/
highest = 0
do player = 1 to n
   highest = max(highest,score.player)
end
say "Highest score is" highest

/*---------------------------------------------------*/
/* Now compute:                                      */
/*  * W, the total number of players that have a score */
/*    equal to HIGHEST                               */
/*  * WINNER.1, WINNER.2 ... WINNER.W, the id-numbers */
/*    of these players                               */
/*---------------------------------------------------*/
w = 0                        /* number of winners    */
do player = 1 to n
   if score.player = highest then do
      w = w + 1
      winner.w = player
   end
end

/*---------------------------------------------------*/
/* announce winners                                  */
/*---------------------------------------------------*/
if w = 1
   then say "The winner is Player #"winner.1
else do
   say "There is a draw for top place.  The winners are"
   do p = 1 to w
      say "     Player #"winner.p
   end
end
say
```

Figure 3-10 (Part 2 of 2). GAME.CMD

Stems and Tails

The *stem* of a compound symbol is the portion up to and including the first period. That is, it is a valid variable name that ends with a period.

The stem is followed by a *tail*, comprised of one or more valid symbols (constants or variables) separated by periods. You can refer to all the variables in an array by using the stem of the array. For example:

```
player. = 0
say player.1 player.2 player.golf      /* displays '0 0 0'  */
```

It is often convenient to set all variables in an array to 0 in this way.

Filling a Two-Dimensional Array

You can have more than one period in a compound symbol. For example, here is the beginning of a program for playing checkers. BOARD is a 2-dimensional array, 8 squares by 8 squares. The squares on the board are called BOARD.ROW.COL and there are 64 of them. Figure 3-11 shows how the men are set at the start of the game.

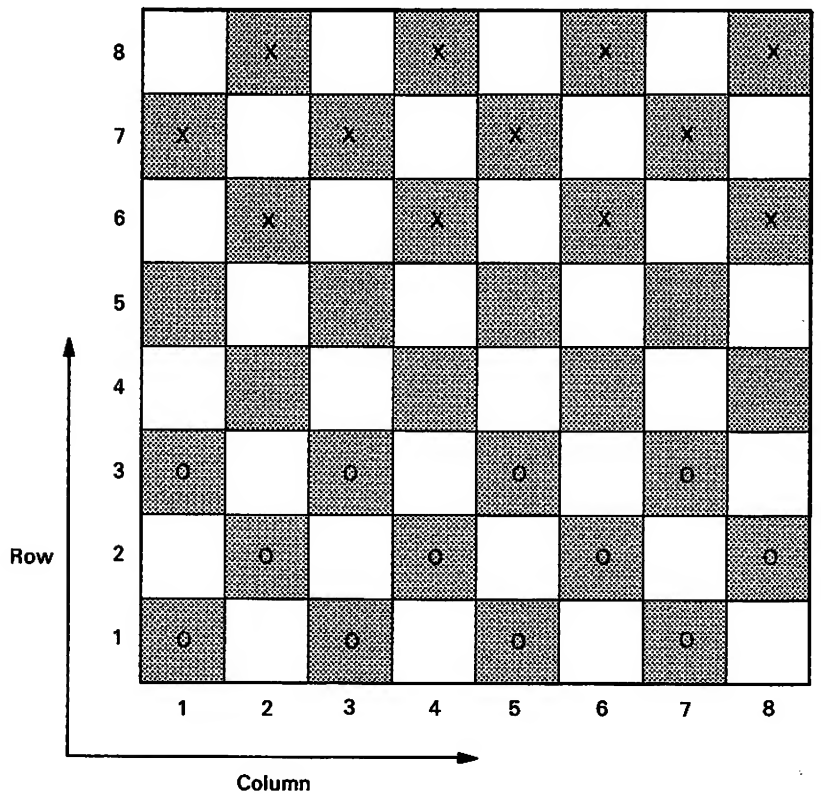


Figure 3-11. Checker Board

Figure 3-12 shows a program that sets the men on the checker board.

```
/* This program simulates a board on which the game of  */
/* checkers can be played.                              */

/* In the internal representation, Red's "men" are      */
/* represented by the character "r" and Red's "kings"   */
/* by the character "R".  Similarly, Black's "men" and  */
/* "kings" are represented by "b" and "B".              */
/*------------------------------------------------------*/
/* Clear the board                                      */
/*------------------------------------------------------*/
board. = " "

/*------------------------------------------------------*/
/* Set out the men                                      */
/*------------------------------------------------------*/
do col = 1 by 2 to 7
   board.1.col = "r"
end
do col = 2 by 2 to 8
   board.2.col = "r"
end
do col = 1 by 2 to 7
   board.3.col = "r"
end
do col = 2 by 2 to 8
   board.6.col = "b"
end
do col = 1 by 2 to 7
   board.7.col = "b"
end
do col = 2 by 2 to 8
   board.8.col = "b"
end
```

Figure 3-12. CHECKERS.CMD

Variables in Programs and Subroutines

Here are some considerations for using variables in REXX programs. For more information about variables, including how programs written in other languages can use REXX variables, see the *REXX Reference*.

Special Variables

REXX has three special variables that are assigned values automatically as needed:

RESULT holds the value set by a RETURN instruction in a subroutine. (See "RETURN Instruction" on page 7-3.)

SIGL holds the line number of the last call to a label. (See "SIGNAL Instruction" on page 7-14.)

RC holds the return code from the last OS/2 command issued. (See "Reading Return Codes" on page 5-9.)

SYMBOL()Function

It is sometimes useful to check whether a symbol has already been used as a name of a variable. To do this, use the SYMBOL() function, SYMBOL(name), where name is the name of the symbol that you want to test. The SYMBOL() function returns:

BAD if name is not a valid symbol

VAR if name has been used as a variable in the program

LIT if the symbol name is a valid variable that has not yet been assigned a value, or if it is a constant symbol (such as a number).

One use of SYMBOL() is to ensure *initialization* of a variable; that is, making certain that the variable is set to a proper starting value before it is used in an operation. For example, you can use SYMBOL() to make sure REXX does not try to add a variable called payment to one named cash until cash has been set to a numeric value.

```
if symbol("CASH") = "LIT" then  cash = 0

cash = cash + payment
```

Put CASH in quotes to test the symbol rather than its value. Notice what happens if the argument of SYMBOL() is not in quotes.

```
cash = 100

say symbol(CASH)        /* displays 'LIT', because the value */
                        /* of CASH is 100 - a constant       */

say symbol("CASH")      /* displays 'VAR', because CASH is   */
                        /* the name of a variable            */
```

Without the enclosing quotes, CASH is treated as a variable and its value is substituted before the function is performed.

PROCEDURE Instruction

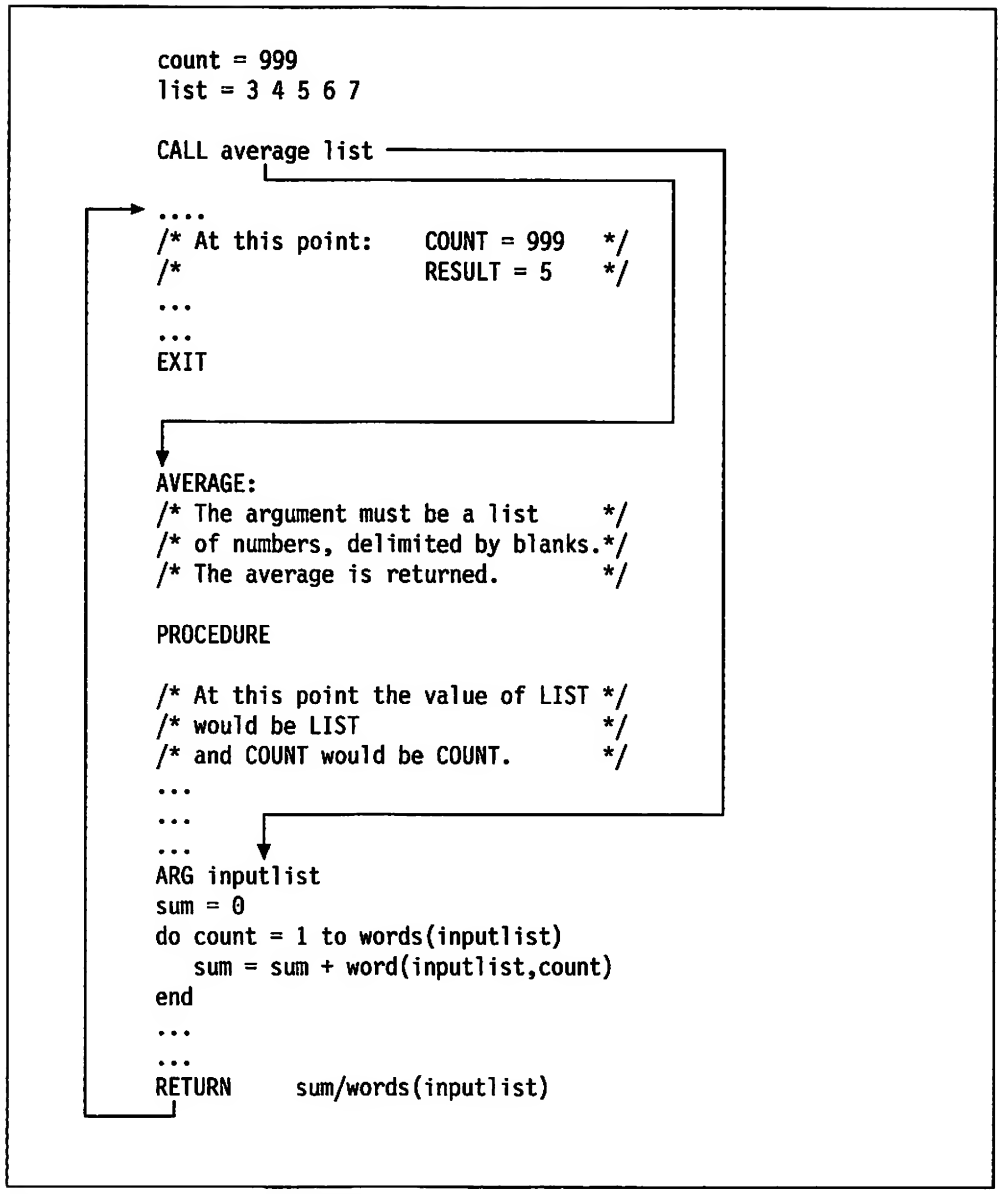
When you are writing a subroutine, you may not know the names of all the variables in the main program. You could check by reading the entire program every time you wanted to create a new name, but this is tedious and prone to error. To delete all variables, for the sake of the subroutine, use the REXX PROCEDURE instruction.

Once this instruction has been processed, new variables can be created that are regarded as different, even if some of them have the same names as variables that existed before the PROCEDURE instruction was processed.

When a RETURN instruction is processed, the new variables are deleted and the original variables are restored.

A PROCEDURE instruction can only be used within an internal routine. It can be used only once and *must* be the first instruction in the routine. For further details on the PROCEDURE instruction, see the *REXX Reference*.

Figure 3-13 shows count being used for two separate purposes.



```
count = 999
list = 3 4 5 6 7

CALL average list

....
/* At this point:    COUNT = 999   */
/*                   RESULT = 5    */
...
...
EXIT


AVERAGE:
/* The argument must be a list     */
/* of numbers, delimited by blanks.*/
/* The average is returned.        */

PROCEDURE

/* At this point the value of LIST */
/* would be LIST                   */
/* and COUNT would be COUNT.       */
...
...
...
ARG inputlist
sum = 0
do count = 1 to words(inputlist)
   sum = sum + word(inputlist,count)
end
...
...
RETURN     sum/words(inputlist)
```

Figure 3-13. PROCEDURE.CMD

PROCEDURE EXPOSE Instruction

To share a limited set of variables between the main routine and the subroutine (leaving all the other variables protected), use `PROCEDURE EXPOSE name [name] [name]...`, where `name` is the name of a variable to be shared.

You can also name a list of variables to be shared. By specifying the stem of an array (`PROCEDURE EXPOSE player.`), you can share all of the variables therein.

For further details, see the discussion of the PROCEDURE instruction in the *REXX Reference*.

Other Types of Data Storage

Variables are the principal means of manipulating data within a REXX program. REXX features other ways to work with data *outside* the program and to share data with other programs, such as:

Queues store sequential data in memory.

Files provide more permanent storage on disk.

Refer to Chapter 10, "Input and Output," for the REXX instructions and functions that work with external data.

Chapter 4. Expressions

An *expression* is a description of information that you want REXX to compute. It can be as simple as adding two numbers or as complex as you want to make it.

Basics

In this chapter:

Basics

- Transforming data
- Operator precedence
- Using functions
- Comparing data
- Using comparisons for program control
- Using expressions in instructions
- Tracing evaluation.

Transforming Data

The process of REXX reading an expression and producing a result is called *evaluating* the expression. REXX has many rules for evaluating expressions.

Expressions are made up of *terms*, the data that is computed, and *operators*, the computations that are performed.

Some examples of REXX expressions are:

```
/*      some arithmetic                 */
say 3 + 2          /* displays "5"      */
say 3 * 2          /* displays "6"      */
say 3 2            /* displays "3 2"    */
say 3||2           /* displays "32"     */


/*      some literal strings            */
say "box" "car"    /* displays "box car" */
say "box""car"     /* displays "box"car" */
say "box"||"car"   /* displays "boxcar"  */


/*      some variables                  */
x = 6
y = 7
say x + y          /* displays "13"     */
x = x + y
say x              /* displays "13"     */
```

Terms

The terms of an expression are the individual pieces of information that you want REXX to work on. The types of terms that make up expressions are:

Numbers Strings that REXX can calculate. REXX recognizes them as constant values. For example:

```
25   3.14159   -6   1989
```

Literal strings Anything within matched quotes REXX accepts as is. For example:

```
"Wednesday"   'C:\PROGRAMS\'   "09 June 89"
```

Variables Symbols that stand for changeable data. When you use a variable in an expression, REXX evaluates it and uses its value as the expression. For example:

```
date = 30            /* stores the number 30          */
                     /* in the variable DATE          */

month = "March"      /* stores the literal string     */
                     /* "March" in the variable MONTH */

say month date       /* displays "March 30" on screen */
```

Function calls These are special computations, some built into REXX, others that you can create. When you use a function call in an expression, REXX performs the function's calculation first, then uses its result as the expression term. For example:

```
say time()             /* displays the current time   */

say substr("REXX",2,1) /* displays "E" on screen;     */
                       /* i.e., 1 character from the  */
                       /* string "REXX" beginning     */
                       /* with the 2nd character      */
```

Basic Operators

The most commonly used operators are those used for arithmetic and *concatenation*. Arithmetic performs adding, subtracting, multiplying, and dividing. Some examples of arithmetic operators are:

```
/* add, subtract, multiply ...        */
say 14 + 5     /* displays "19"       */
say 14 - 5     /* displays "9"        */
say 14 * 5     /* displays "70"       */


/* ... and three ways to divide       */
say 14 / 5     /* displays "2.8" -    */
               /* normal division     */


say 14 % 5     /* displays "2" - the  */
               /* integer result only */


say 14 // 5    /* displays "4" -the   */
               /* remainder only      */
```

For more information about using numbers in REXX, see Chapter 9, "Arithmetic."

Concatenation joins strings together with:

- A single blank, if you leave one or more blanks between the terms of an expression
- No intervening blanks, if you put the terms together. This is called *abuttal.* For abuttal to work, REXX must be able to recognize the terms as separate.
- No intervening blanks, if you use the concatenation operator ¦¦ (two vertical bars). Use this operator where abuttal does not work (such as with two variables) or to state exactly how you want the strings joined.

Some examples of concatenation are:

```
say "slow"    "coach"  /* displays 'slow coach' */

adjective = "slow"
say adjective  "coach" /* displays 'slow coach'          */

say adjective"coach"   /* displays 'slowcoach'           */
                       /* (using abuttal)                */

say "slow"¦¦"coach"    /* displays 'slowcoach'          */


say 4 5                /* displays '4 5'                */
say 4¦¦5               /* displays '45'                 */

tens = 4
units = 5
say tensunits          /* displays 'TENSUNITS' (abuttal     */
                       /*  produces a new symbol)           */

say tens¦¦units        /* says '45'                */
say (4¦¦5) / 3         /* displays '15'                 */
```

Concatenation works with numeric as well as non-numeric strings. In the previous example, parentheses are used to force REXX to concatenate the 4 and the 5 *before* dividing the result by 3.

Operator Precedence

Without parentheses in the previous example, REXX would have performed the division first, then the concatenation. It would divide 5 by 3 (result 1.6666667) and then join the 4 in front. For example:

```
say 4¦¦5 / 3           /* displays '41.6666667' */
```

REXX gives *precedence*, or priority, to division over concatenation. This means that REXX has a set of built-in rules for the order in which operations are performed.

Basically, REXX evaluates an expression by reading it from left to right. Before it does, it takes into account:

- Any parentheses you have used.
- The built-in operator precedence of REXX, which means some operations get higher priority than others, no matter where they are in the expression.

Multiplication and division come before addition and subtraction; addition and subtraction before concatenation. See page 4-10 for a listing of REXX operator precedence.

You can always use parentheses to override the precedence rules of REXX. It is a good practice to use parentheses because they help make your program more readable.

Using Functions

For more extensive computations, REXX has built-in *functions*. A function always *returns* (produces) a value. This value is represented in an expression by a symbol referred to as a *function call*. All function calls consist of a name, followed by parentheses. There is no space between the name and the first parenthesis. For example:

```
length("Madison")
```

- The value you want the function to work on is called the function's *argument*. In this example, the argument is the literal string "Madison".
- The value a function call produces is its *return value*. The return value depends on what you put inside the parentheses. In this example, the LENGTH() function returns the value 7, the number of characters in Madison.

The instruction, `say length("Madison Avenue")` displays the return value 14.

The argument of a function can itself be an expression. For example, the function ABS() returns the absolute (positive) value of a given number. Given an arithmetic expression as an argument, ABS() first evaluates the expression and gets the result. In the following example, the ABS() function calculates the result -98 and returns the absolute value of that result, which is 98.

```
say abs(2 - (50 * 2))      /* displays '98' */
```

If a function allows for or requires more than one argument, the arguments are separated by commas. For example:

```
say copies("=",80)  /* displays 80 "=" characters */
```

Some functions need no arguments. For example, the DATE() function gets the date from your system clock. It has several optional arguments.

```
say date()          /* displays '15 Mar 89'   */
say date(w)         /* displays 'Wednesday'   */
say date(s)         /* displays '19890315'    */
```

The DATATYPE() function reports the type of data an expression evaluates as:

```
str1 = "ABC"        /* a character string     */
str2 = "12"         /* a numeric string       */
say datatype(str1)  /* displays 'CHAR'        */
say datatype(str2)  /* displays 'NUM'         */
```

You can include a function call anywhere within an expression. REXX performs the function's computation and then substitutes the return value for the function call *before* it evaluates the entire expression.

For more information about functions, see Chapter 7, "Program Structure."

Refer to the *REXX Reference* for descriptions of the REXX built-in functions.

Comparing Data

Comparisons *test* data rather than manipulate it. The following is an example of a comparison in a program (see Figure 2-1 on page 2-1).

```
if who = "" then say...
```

What happened next in the program depended on the test: was the variable who empty?

Comparisons are performed using the following operators:

Operator	Meaning
=	Equal
\=	Not equal
<> or ><	Not equal
>	Greater than
\>	Not greater than
<	Less than
\<	Not less than

The comparison operators can be combined so that >= stands for *greater than or equal* and \>= means *not greater than or equal* (same as <).

The backslash character (\) negates a comparison operator, turning *equal* to *not equal* and so on. This character is also used for *not* in expressions generally. (See NOT Operator (\) on page 4-16.)

The equal sign (=) can have different meanings in REXX depending on its position in a clause. For example:

```
amount = 5      /* The variable AMOUNT gets the value 5     */

say amount = 5  /* Compare the value of AMOUNT with 5       */
                /* If they are the same, displays '1'       */
                /* Otherwise, displays '0'                  */
```

The rule is that a clause beginning with symbol = ... is an *assignment*. An equal sign anywhere else in a clause usually stands as a comparison operator. There are exceptions. The equal sign has a special use in the PARSE instruction, and it is a character, not an operator, in a comment or literal string.

Testing for True or False

The result of a comparison expression is either *true* (1) or *false* (0). For example:

```
/* some comparisons */
say 5 = 5                    /* displays '1'       */
say 5 <> 5                   /* displays '0'       */
say 5 = 4                    /* displays '0'       */
say 2 + 2 = 4                /* displays '1'       */
say 2 + 2 = 5                /* displays '0'       */

howmuch = 2 + 3              /* assigns the sum    */
                             /* of 2 and 3 to the  */
                             /* variable HOWMUCH   */

say "apples" = "oranges"     /* displays '0' */

fruit = "oranges"            /* assigns the string */
                             /* "oranges" to the   */
                             /* variable FRUIT     */

say fruit = "apples"         /* displays '0'       */

say fruit = "oranges"        /* displays '1'       */

say howmuch fruit          /* displays "5 oranges" */

say howmuch fruit = "4 oranges"  /* displays '0'  */
say howmuch fruit = "5 plums"    /* displays '0'  */
say howmuch fruit = "5 oranges"  /* displays '1'  */
```

Combining Expressions

It is often useful to combine two or more comparisons to find a single true-or-false value. You can have REXX evaluate a *set* of comparisons and compute an overall value of 1 or 0 for the set. You can combine comparisons so that REXX evaluates them as:

- *True* (1) only if *all* of the comparisons in the set evaluate as 1, but *false* (0) if *any one or more* of them evaluates as 0. This is called an AND condition. To combine comparisons this way, use the AND (&) operator. For example:

  ```
  fruit = "grapes"
  howmuch = 10
  say fruit = "apples" & howmuch = 10   /* displays '0'   */
  say fruit = "grapes" & howmuch = 5    /* displays '0'   */
  say fruit = "grapes" & howmuch = 10   /* displays '1'   */
  ```

  Only in the third example are both of the comparisons *true*, so only, then, is the combined result evaluated as 1.

- *True* (1) if *any one or more* of the comparisons in the set evaluate as 1, and *false* (0) only if all the comparisons evaluate as 0. This is called an OR condition. To combine comparisons this way, use the OR (|) operator. For example:

```
fruit = "grapes"
howmuch = 10
say fruit = "apples" | howmuch = 10   /* displays '1'   */
say fruit = "grapes" | howmuch = 5    /* displays '1'   */
say fruit = "grapes" | howmuch = 10   /* displays '1'   */
say fruit = "apples" | howmuch = 5    /* displays '0'   */
```

  Both comparisons, individually, had to evaluate as 0 in order for their combined value to be 0.

Using Comparisons for Program Control

REXX has a set of instructions that control the program and choose the action a program is to take in a given situation. That situation is determined by comparisons.

Instructions, such as `IF expression THEN ... ...`, must be given an expression that computes to 0 or 1. In the example of IF...THEN..., a result of 1 means the clause following THEN is processed; a result of 0 means that it is not processed.

The following two examples give the same result.

```
ready = "YES"
⋮
if ready = "YES" then ...
⋮
```

or

```
ready = 1
⋮
if ready then ...
⋮
```

You can use whichever form you prefer.

For more information about how comparisons can control a program's processing, see Chapter 6, "Program Control."

Using Expressions in Instructions

The *REXX Reference* provides an alphabetical listing of all REXX instructions and their syntax. Where the syntax of an instruction calls for or allows an *expression*, you can use any of the rules described here.

Tracing Evaluation

If your program produces unexpected results, the problem may be that an expression was mis-stated. When your program will not run, use the TRACE instruction. TRACE displays how REXX evaluates expressions while the program is actually running. The options most often used are:

TRACE Intermediates	Displays the *immediate* result of each operation.
TRACE Results	Displays only the final result of each expression, after it has been evaluated.

When a TRACE instruction is being interpreted, the first letter of the option determines the type of tracing that is switched on; the rest of the word is ignored.

For example, to TRACE intermediate results for an expression, you could write:

```
TRACE I
... expression
```

Figure 4-1 shows a program that uses the TRACE I instruction.

```
/* Example: to show how an expression      */
/* is evaluated, operation by operation    */
x = 9
y = 2
trace I                 /* Switch on tracing. */
if x + 1 > 5 * y then   /* If the comparison  */
                        /* evaluates as '1'   */
say "x is big enough."  /* ...display this.   */
```

Figure 4-1. TTRACE.CMD

The following is displayed on the screen when you run the program.

```
[C:\] ttrace
     6 *-* If x + 1 > 5 * y
       >V>   "9"
       >L>   "1"
       >O>   "10"
       >L>   "5"
       >V>   "2"
       >O>   "10"
       >>>   "0"
[C:\]
```

The symbols mean:

- An instruction is being traced. The number on the left is the line number in your program.

>V> Value of a Variable.

>L> Value of a Literal.

>O> Result of an Operation.

>>> The final result of the evaluation.

Figure 4-1 on page 4-8 shows that the final result is *false* (0). Because the IF expression is false, the THEN clause is not processed.

You may not need a trace of every intermediate result. For example, to display only the final evaluation results, use TRACE results.

```
TRACE R
... expression
```

Figure 4-2 shows a program that uses the TRACE R instruction.

```
/* Example: to show how an expression is evaluated,     */
/* operation by operation using TRACE R                 */
x = 9
y = 2
trace R                 /* Switch on tracing. */
if x + 1 > 5 * y then   /* If the comparison  */
                        /* evaluates as '1'   */
say "x is big enough."  /* ...display this.   */
```

Figure 4-2. RTRACE.CMD

The following is displayed on the screen when you run the program.

```
[C:\] rtrace
     6 *-* if x + 1 > 5 * y
       >>>   "0"
[C:\]
```

Here too, the symbol >>> indicates the final result. Again, the final result is *false* (0). Because the IF expression is false, the THEN clause is not processed.

Summary

This completes "Basics" in this chapter. You have learned how to:

- Use basic REXX arithmetic
- Join strings by concatenation
- Test data using comparison operators
- Use trace evaluation while a program runs.

"Advanced Topics" in this chapter discusses:

- Operator precedence
- Using parentheses
- Manipulating strings with functions
- More about comparisons.

To continue with "Basics," go to page 5-1.

Advanced Topics

In this chapter:

Advanced Topics

- Precedence
- Using parentheses
- More about numbers
- Concatenation
- Substring functions
- Comparisons
- Translating and converting data.

Precedence

When evaluating an expression, REXX reads the operations from left to right. But some operators are given a higher precedence (priority) than others and are processed first regardless of their position. The complete order of precedence of the operators by group (highest priority at the top) is:

Prefix operators	\ − +
Powers	**
Multiply and divide	* / % //
Add and subtract	+ −
Concatenation with/without blank	" " || abuttal
Comparison operators	== = \== \= > < >> << >< <> >= \< >>= \<< <= \> <<= \>>
Logical *and*	&
Logical *or* (inclusive/exclusive)	| &&

From this list, you can determine the sequence of operations for any expression. For example:

```
Say 3 + 2 * 5          /*    displays '13'  */
```

Because multiply (*) has a higher priority than add (+), the multiply operation is done before the operation on its left. Similarly, because add (+) has a higher priority than concatenate (blank), the add operation is done before the concatenate operation. For example:

```
Say 3 2+2 5            /* displays '3 4 5' */
```

Using Parentheses

You can use parentheses to force evaluation in a different order since expressions inside parentheses are evaluated first. For example the value of:

```
6 - 4 + 1      is  3.
6 -(4 + 1)     is  1.
3 + 2||2 + 3   is  55.
3 +(2||2)+ 3   is  28.
```

Test Yourself

What is the value of:

1. 4 + 20 "tailors"
2. 24 = 4 + 20
3. "eggs" = "eggs" & 2*2 = 4
4. 3 / 2*5
5. 3 || 7+7
6. 3(2+2)
7. (2+2)3.

Answers:

1. 24 tailors (add before concatenate)
2. 1 (add before comparison)
3. 1 (comparison before AND, multiply before AND, comparison before AND)
4. 7.5 (operators that have the same priority are processed left to right)
5. 314 (add before concatenate)
6. calls the function 3 with the argument 4 (or gives a syntax error if 3 does not exist)
7. 43 (evaluate expression in parentheses first; then do the abuttal).

More about Numbers

REXX regards everything in a program as a string to be evaluated. REXX treats certain strings as numbers that can be calculated, such as:

- A number begins with a digit, decimal point, or sign (+ or −).
- Powers of 10 are indicated by E. These are called *floating point* numbers.
- Numbers may be in quotes, and spaces are allowed around the plus or minus sign.

For more information about valid numbers, see "About REXX Numbers" on page 9-1.

Concatenation

The three concatenation operations are:

- Leave one or more spaces between the terms. REXX joins the terms with a single blank.
- Put the terms together with no space. You can do this so long as REXX can recognize the terms as separate
- Use the || operator to join the terms without a blank.

Substring Functions

The following two substring functions are useful when working with strings.

Getting Pieces of Strings

To select a part of a string (a *substring*), use the SUBSTR() function. For example:

```
substr(string,n)
substr(string,n,length)
```

where:

`string` is the string from which the substring is taken.

`n` is the position of the character in `string` that is the first character of the substring. (Characters in a string are numbered 1,2,3,...)

`length` (optional); is the number of characters in the substring. If you omit it, the rest of `string` (from the nth position to the end) is returned.

An example using the SUBSTR() function is:

```
S = "reveal"
say substr('revealing',1,6)    /* displays 'reveal'   */
verb = substr('revealing',1,6) /* stores 'reveal' */
                               /* in  a variable  */

say substr(verb,2)      /* displays 'eveal' */
say substr(verb,2,3)    /* displays 'eve'   */
say substr(verb,3,4)    /* displays 'veal'  */
```

Finding Lengths of Strings

To find out the length of the result of any string or string expression, use the LENGTH() function, `length(n)`, where `n` is the number of characters in the string. For example:

```
verb = "reveal"
say length(verb)      /* displays '6'    */
```

Figure 4-3 shows a program that uses these two functions. The program checks a file name typed by a user.

```
/* Checking a file name */
say "Type a file name"
pull fname"."ext          /* Pull reads the file name   */
                          /* up to the period (if an   */
                          /* extension is entered)     */
if length(fname) > 8
then
   do
      fname = substr(fname,1,8)
      say "The file name you typed was too long. ",
          fname "will be used."
   end
```

Figure 4-3. CHKFNAME.CMD

For more information about substrings, see "String Functions" on page 8-13.

Parsing

Another way to manipulate and analyze string expressions is by *parsing*. See Chapter 8, "Parsing."

Comparisons

Comparisons are performed using these operators:

=	(equal)
\= **or** < > **or** > <	(not equal)
>	(greater than)
\>	(not greater than)
<	(less than)
\<	(not less than).

Comparing Numbers

Use comparison operators in an expression to compare the terms. The result is 1 if the comparison is true; 0 if the comparison is false.

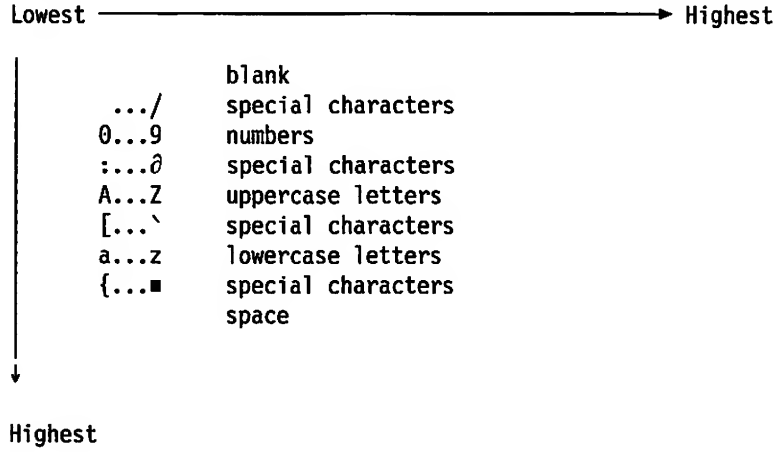
For strings that are numbers, on which REXX can perform arithmetic, comparisons work differently. If both the terms being compared are numbers, comparison is numeric, rather than character-by-character. For example the value of:

5 > 3 is 1 (true)
2.0 = 002 is 1 (true)
3E2 < 299 is 0 (false).

Comparing Characters

Comparing characters is performed by using one of two methods, *normal comparison* or *strict comparison*. Normal comparison ignores leading and trailing blanks and compares character-by-character. Strict comparison compares character-by-character including any blanks.

The value of a character is less than another character according to this sequence of lowest to highest value.



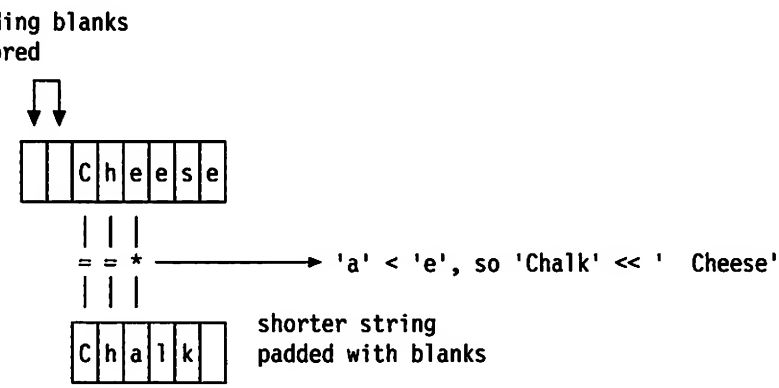
Note: Special characters are mixed between the previously listed categories of numbers and letters as shown. Only unaccented characters and numbers are in numerical order in any of the code pages supported by the OS/2 program. All

characters are compared through pure binary sorting according to their order in the code page you are using. For more information on the special characters for the primary code page you are using, see *Keyboard Layouts.*

Normal Comparison

If either of the terms is not a number, leading and trailing blanks are ignored. The shorter string is padded on the right with blanks and then the strings are compared from left to right, character-by-character. If the strings are not equal, the first pair of characters that do not match determine the result.

For example, if " Cheese" is compared with "Chalk ":



Test Yourself

1. What is the value of each of the following expressions?
   a. "3" < "five"
   b. "Kilogram" < "kilogram"
   c. "a" > "#"
   d. "q" > "?"
   e. "9a" > "9"
   f. "?" > " "
2. What is displayed on the screen when the program shown in Figure 4-4 is run?

```
/* A fair comparison */
say "Apples" = "Apples"
```

Figure 4-4. FAIR.CMD

Answers:

1. All are 1 (true).
2. The following is displayed on the screen; because "Apples" is equal to "Apples", the result is 1 (true).

```
[C:\] fair
1
[C:\]
```

Strict Comparisons

By using strict-comparison operators, we can specify character-by-character comparisons, with no padding of either of the strings. They do not try to perform numeric comparisons because they test for an exact match between the two strings.

To find out whether two strings are exactly equal, use the double-equal sign (==) operator. For example:

The value of "cookies" = " cookies" is 1 (true)

The value of "cookies" \= " cookies" is 0 (false)

The value of "cookies" = = " cookies" is 0 (false)

The value of "cookies" \= = " cookies" is 1 (true)

Blanks are *lower* in the ASCII sequence than letters. Strict comparison compares the leading blanks in its evaluation.

To find out whether two strings are exactly greater than or exactly less than, use the double-greater-than (>>) and double-less-than (<<) operators. A character is *less than* another character if it comes earlier in the sequence, see "Comparing Characters" on page 4-13. For example the value of:

"cookies" > > "carrots" is 1 (true)

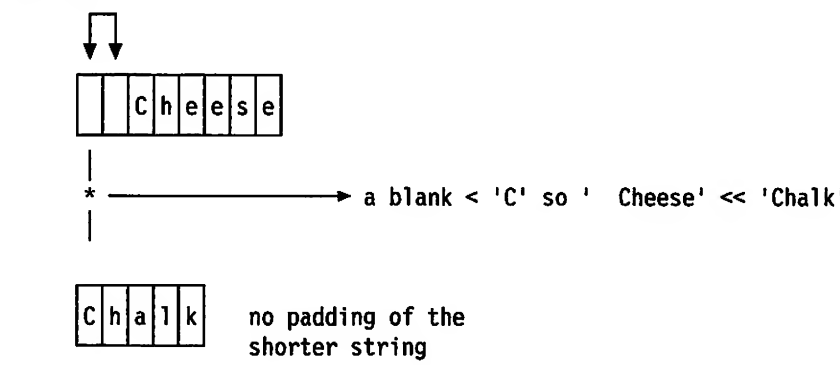
"$10" > > "nine" is 0 (false)

"steak" < < "fish" is 0 (false)

" steak" < < "steak" is 1 (true).

Since the blank is lower in the sequence of characters, " steak" is strictly less than "steak".

See the difference now when we strictly compare " Cheese" and "Chalk":



Strict-comparison operators are especially useful when you want to compare strings for leading and trailing blanks.

Strict comparison is not usually applied to numeric strings, because the reason for such a comparison is generally to compare values rather than characters. Given that x = "2" and y = "+2", the value of:

x = y is 1 (true)

x \= y is 0 (false)

x == y is 0 (false)

x \== y is 1 (true).

Strict comparison is also useful if you want to check for nonsignificant zeroes or exponential notation. For example, the value of:

32.000 = 32 is 1 (true)

32.000 == 32 is 0 (false)

000000 = 0E0000 is 1 (true)

000000 == 0E0000 is 0 (false).

Boolean Operators

The logical or *Boolean* operators modify and combine expressions. The Boolean operators are:

- NOT Operator (\)—A *not* operator (\) placed in front of a term changes its value from *true* to *false* or from *false* to *true*.

  A NOT operator (\) reverses the result of any comparison it precedes. An expression that REXX otherwise evaluates as 1 is changed to 0 when you put the NOT operator in front of it. Similarly, an expression that REXX would otherwise evaluate as 0 is evaluated as 1 when preceded by a NOT operator. For example:

```
say \ 0                    /* displays '1'             */
say \ 1                    /* displays '0'             */
say \ 2                    /* returns a syntax error   */

say \ (3 = 3)              /* displays '0'             */


fruit = "oranges"          /* assigns "oranges" to      */
                           /* the variable FRUIT       */
say fruit = "oranges"      /* displays '1'             */
say fruit = "apples"       /* displays '0'             */
say \(fruit = "apples")    /* displays '1'             */
say \(fruit = "oranges")   /* displays '0'             */
```

  To combine comparisons, to get the overall true-or-false value of more than one condition, use the logical operators AND and OR.

- AND Operator (&)—To write an expression that is only true when every one of a set of comparisons is true, use the AND (&) operator. For example:

```
If ready = "YES" & steady = "RIGHT"
then say "GO"
```

  This means that if ready has a value of YES and steady has a value of RIGHT, then say GO. Otherwise, do nothing.

- Inclusive OR Operator (¦)—The single vertical bar (¦) is an *inclusive* OR. It combines comparisons so that the whole expression evaluates as 1 (true) if *any* of the comparisons are true.

To write an expression that is true when at least one of a set of comparisons is true, use the inclusive OR (¦) operator. For example:

```
If ready = "YES" ¦ steady = "RIGHT"
then say "GO"
```

This means that if either ready has a value of YES or steady has a value of RIGHT, or both, then say GO. Otherwise, do nothing.

- Exclusive OR Operator (&&)—The double ampersand (&&) is an *exclusive* OR. It combines comparisons so that the expression evaluates as 1 (true) when one and only one of the comparisons is true.

```
city = 'NEW YORK'
state = 'NJ'
local = 'NO'
if city = 'NEW YORK' && state = 'NJ' then
local = 'YES'
say local                          /* displays 'NO' */
```

Test Yourself

What is displayed on the screen, when the program shown in Figure 4-5 is run?

```
/* Example: comparing numbers */
dozen = 12
score = 20
say score = dozen + 8

/* Using the AND operator */
say dozen = 12 & score = 21

/* Using the OR operator */
say dozen = 12 ¦ score = 21
```

Figure 4-5. MEASURES.CMD

Answer: The following is displayed on the screen when you run the program.

```
[C:\] measures
1
0
1
[C:\]
```

For the expression using the AND operator:

- The first comparison (dozen = 12) produces the result 1 (true).
- The second comparison (score = 21) produces the result 0 (false).

The result of the AND operation is 0 (false). The AND operation evaluates as 1 (true) *only* when both comparisons evaluate as 1.

For the expression using the OR operator:

- The first comparison (dozen = 12) produces the result 1 (true).
- The second comparison (score = 21) produces the result 0 (false).

The result of the OR operation is 1 (true). The OR operation evaluates as 1 (true) when *at least one* of the comparisons evaluate as 1.

Translating and Converting Data

REXX functions can translate data from one character set to another. Translation is especially useful when a REXX program must process output that is in binary or hexadecimal form. If you are a beginning programmer, you may not need these functions. However, you may want to read this discussion if you want to use REXX with your printer or with other programs.

For more information on how REXX works with external data in files and queues, refer to Chapter 10, "Input and Output."

For more information about the following functions, refer to "Functions" in the *REXX Reference*.

Number Systems

The OS/2 program uses a standard character set, called the ASCII (American Standard Code for Information Interchange) set, to represent information. In ASCII, each character has a unique number that may be represented in different ways. For example, the character ? has the value 63.

Computers operate in *binary* terms such as on and off, yes and no, true and false. They regard numbers in the same way. The decimal number 63 in a binary numbering system is `00111111`. Each digit stands for a power of two. In this example the binary number stands for 32+16+8+4+2+1, or 63. That is how a computer sees a question mark. Every character is rendered as a *byte*: a set of eight binary digits, each having a value of `0` or 1. Binary numbers are difficult for people to read.

Programmers use another numbering system, based on 16 digits, called *hexadecimal* or *hex*, for short. Each digit of a hex number represents four binary digits. Therefore, a byte can be represented by just two digits. The 16 possible hex digits are:

```
0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F (hex)
0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 (decimal)
```

The hexadecimal equivalent of decimal 63 is `3F`. That is 3*16 plus 15*1, or 63, which translates to the question-mark (?) character.

Note: For some decimal or hexadecimal numbers, the ASCII character is different, depending on the code page that you are using (see *Keyboard Layouts*).

In addition to ASCII and decimal numbers, REXX also accepts strings expressed in these forms:

Binary If a string is to be expressed as one or more binary numbers, enclose it in matching quotes and put the letter `B` (uppercase or lowercase) immediately following the closing quote. Spaces within the string are allowed only to separate digits into groups of four or eight.

Hexadecimal If a string is expressed in hex, place the letter X (uppercase or lowercase) immediately following the closing quote. Spaces within the string are allowed only to separate the digits into pairs. For example:

```
say 37             /* displays '37' (decimal number)          */
say "%"            /* displays '%' (ASCII character 37)       */
say '25'x          /* '37' in hex; says '%' (ASCII 37)     */
say '0010 0101'b   /* '37' in  binary; says '%' (ASCII 37) */
```

Note: If you use x or b as variables, REXX gives precedence to their use as number-base indicators. For example:

```
x = 'mno'       /* assign a string to variable X */
say '3F'x       /* displays the character '?'    */
                /* ... not '3Fmno'               */
```

Using Functions to Convert Data

The following REXX built-in functions convert numbers from one form to another or convert character data to its numeric form.

B2X() Binary to hexadecimal
C2X() ASCII to hexadecimal
C2D() ASCII to decimal
X2B() Hexadecimal to binary
X2C() Hexadecimal to ASCII
X2D() Hexadecimal to decimal
D2C() Decimal to ASCII
D2X() Decimal to hexadecimal.

The following function calls represent the conversions they perform.

2 means translate to
B means binary
C means characters (that is, ASCII)
X means hexadecimal.
D means decimal.

Thus the value of:

B2X(0011 1111) is 3F
X2B(3F) is 00111111
C2X(?) is 3F
X2C(3F) is ?
C2D(?) is 63
D2C(63) is ?
D2X(63) is 3F
X2D(3F) is 63.

All these functions accept strings more than 1 byte long.

Note: For some decimal or hexadecimal numbers, the ASCII character is different, depending on the code page that you are using (see *Keyboard Layouts*).

Binary numbers can only be translated to and from hexadecimal numbers. To translate binary into other forms, use nesting. For example, the value of:

X2C(B2X(0011 1111)) is ?
X2B(D2X(63)) is 00111111.

Figure 4-6 shows a table of conversion functions.

To Change *n* From	To Binary	To ASCII	To Hex	To Decimal
Binary		X2C(B2X(*n*))	B2X(*n*)	X2D(B2X(*n*))
ASCII	X2B(C2X(*n*))		C2X(*n*)	C2D(*n*)
Hex	X2B(*n*)	X2C(*n*)		X2D(*n*)
Decimal	X2B(D2X(*n*))	D2C(*n*)	D2X(*n*)	

Figure 4-6. Conversion Table

For complete descriptions of these functions, refer to the *REXX Reference*.

Chapter 5. Commands

Commands are instructions that REXX passes to another *environment*. You can use a program written in REXX to control other programs, including the operating system (OS/2 program) for your computer.

If you use an application program, such as a word processor, that is controlled by *subcommands* (the application commands, typed at a user prompt), you can use REXX to create *macros*, programs that issue a series of subcommands to an application. You can, in effect, create your own commands.

You may now be using the OS/2 Batch Facility to automate the OS/2 tasks. Using a REXX program instead, with its variables, its control structures, and its math and parsing capabilities, can make these procedures much more powerful.

Basics

In this chapter:

Basics

- Environment
- From REXX to the OS/2 Program
- From the OS/2 Program to REXX.

Environment

Environment is a term used to describe the workspace that the OS/2 program and certain applications create for themselves. You may already be familiar with the idea of an environment with respect to the OS/2 command set, which defines (or displays) various settings such as default paths and files.

Using REXX, the term environment refers not so much to individual features of the OS/2 program or other applications, but rather to the way they operate generally. REXX does not impose an environment of its own. Instead, it operates entirely within the environment (default environment) from which it is called. This could be the environment of the OS/2 program or of any application program that can use REXX as a macro or *scripting* language.

The basic rule is that whatever REXX cannot process, it evaluates and then passes the result to the default environment.

The concept of environment becomes important when you use a REXX program to issue commands to other programs. Fortunately, the REXX language makes this easy. Starting your REXX programs from the OS/2 command line makes the OS/2 program the default environment for REXX commands.

From REXX to the OS/2 Program

There are three ways to issue a command to the OS/2 program. You can:

- Let REXX evaluate part or all of a clause as an expression. The resulting string is automatically passed to the OS/2 program.

- Enclose the entire clause in quotes. That renders it a literal string to be passed to the OS/2 program.
- Send a command explicitly to the OS/2 program by using the ADDRESS instruction.

Issuing a Command Expression

REXX processes your program one clause at a time. It examines each clause to determine if it is:

- A keyword instruction, such as:

  ```
  say "Type total number"
  ```

  or

  ```
  pull input
  ```

- A variable assignment (any valid symbol followed by an equal sign), such as:

  ```
  price = cost * 1.2
  ```

- A label for calling other routines (see "Subroutines" on page 7-1)
- A null (empty) clause.

If the clause is none of the above, REXX evaluates the *entire clause as an expression* and passes the resulting string to the OS/2 program.

If the string is a valid OS/2 command, then the OS/2 program processes it as though you had typed the string at the command prompt and pressed the Enter key.

Figure 5-1 shows a REXX clause that uses the DIR command in the OS/2 program to display a list of files in the current directory.

```
/* display current directory */
say "DIR command via REXX"
dir
```

Figure 5-1. DIRREX.CMD

The clause `dir` is not a REXX instruction or a label, so REXX evaluates it and passes the resulting string to the OS/2 program. The OS/2 program recognizes the string DIR as one of its commands and processes it. This is also true if the PATH command is used to display the current search path for processible files (see Figure 5-2).

```
/* display current path      */
say "PATH command via REXX"
path
```

Figure 5-2. PATHREX.CMD

REXX evaluates the clause `path` and passes the string PATH to the OS/2 program as a command.

Figure 5-3 shows a program using the DIR and PATH commands. The PAUSE command is added to wait for the user to press a key before issuing the next instruction or command. Also, borders were added.

```
/*    Issue DIR and PATH commands to OS/2      */

say copies('=',40)    /* display line of "="'s */
                      /* for a border          */

dir                   /* display listing of    */
                      /* the current directory */

pause                 /* pauses processing and */
                      /* tells user to "Press  */
                      /* any key to continue." */

say copies('=',40)    /* display line of '='   */
path                  /* display the current   */
                      /* PATH setting          */
```

Figure 5-3. DP.CMD

The following is displayed on the screen when you run the program.

```
[C:\]dp
========================================

 The volume label in drive C is OS2.
 Directory of C:\EXAMPLES

.            <DIR>     10-16-88  12:43p
..           <DIR>     10-16-88  12:43p
EX4_1    CMD     nnnn  10-16-88   1:08p
DEMO     TXT      117  10-16-88   1:10p
     4 File(s)   12163072 bytes free
Press any key when ready . . .

========================================
PATH=C:\OS2;C:\OS2\UTIL;C:\OS2\INSTALL
[C:\]
```

Echoing of OS/2 Commands:

When your REXX program issues an OS/2 command, REXX passes the command to the OS/2 command handler for processing. This processing includes displaying the command on the screen (*echoing*). Commands are also echoed in old-style (non-REXX) .CMD files.

This echo can be distracting. To suppress it, issue the ECHO OFF command from your REXX program. This tells the OS/2 program to suppress all echoing for the remainder of your REXX program. To suppress echoing for one command, put an at sign (@) in front of the command.

Note: For simplification, the examples in this book do not echo commands.

Issuing a Command to Call a .CMD File

If you are issuing a command to have the OS/2 program run one of its built-in commands or other programs, you can call it by name as previously shown. However, to have the OS/2 program run another .CMD program from your REXX program, you must call it using a CALL instruction instead of calling it by name. There are two kinds of calls you can use:

- The REXX CALL instruction
- The OS/2 CALL command.

The REXX CALL instruction calls other REXX programs. To call a REXX program named `mysub1`, you could write the CALL instruction, `call mysub1`. REXX recognizes the CALL instruction, handles the call, and processes `mysub1` as a REXX program.

The REXX CALL instruction does not work to call a non-REXX .CMD file. Instead, you would use the OS/2 CALL command. To call a non-REXX .CMD program named `mysub2`, you could write the CALL instruction, `"call mysub2"`. REXX evaluates the expression and passes it to the OS/2 command handler for processing. The command handler recognizes the CALL command and processes `mysub2` as a .CMD program.

Using Variables

Figure 5-4 shows a program that builds a command string using a variable for user input. It prompts the user to type a file name and then builds a variable to hold the TYPE command and the input file name.

To have REXX issue the command to the operating system, put the command string on a line by itself. REXX evaluates the string and tries to issue it as a command, which is passed to the OS/2 program.

```
/* Issue TYPE command with an input file name  */

/* prompt the user for a file name          */
say "Type a file name:"

/* assign the response to variable FILENAME */
pull filename

/* build a command string by concatenation  */
commandstr = "TYPE" filename

/* If the user typed "demo.txt":            */
/* Now the variable COMMANDSTR contains     */
/* the string "TYPE DEMO.TXT" and so...     */

commandstr          /* ...REXX passes the  */
                    /* string on to OS/2   */
```

Figure 5-4. SHOFIL.CMD

The following is displayed on the screen when you run the program.

```
[C:\]shofil
Type a file name:
?
demo.txt

This is a sample text file. Its sole
purpose is to demonstrate how OS/2
commands can be issued from REXX
programs.

[C:\]
```

Using Quotes

The rules for forming a command from an expression are exactly the same as those for forming expressions. Be careful of symbols that have meanings for both REXX and OS/2 programs. To determine how REXX evaluates a command when the command name and a variable name are the same, use the program DIRREX.CMD (see Figure 5-1 on page 5-2) and assign a value to the symbol DIR as shown in Figure 5-5.

```
/* assign a value to the symbol DIR */
say "DIR command via REXX"
dir = "echo This is not a directory!"

/* pass the evaluated variable to OS/2 */
dir
```

Figure 5-5. DIRREX2.CMD

Now `dir` is a variable with the string, `"echo This is not a directory!"`, as the assigned value.

The following is displayed on the screen when you run the program.

```
[C:\]dirrex2
DIR command via REXX:
This is not a directory!
[C:\]
```

REXX evaluates a literal string, a string enclosed in matching quotes, exactly as it is found. To ensure that a symbol in a command is not evaluated as a variable, enclose it in matching quotes as shown in Figure 5-6.

```
/* assign a value to the symbol DIR */
say "DIR command via REXX"
dir = "echo This is another string now!"

/* pass the literal string "dir" to OS/2 * /
"dir"
```

Figure 5-6. DIRREX3.CMD

The result displayed on the screen is now a directory listing.

The best way to ensure that REXX passes a string to the OS/2 program as a command is to enclose the entire clause in quotes. This is especially important when you use OS/2 symbols that REXX uses as operators.

If you were trying to erase a set of files by *wildcard pattern*, the clause, `delete *.bak`, would result in a syntax error. REXX would read the * as its multiplication operator and you cannot start a variable with a period.

The same thing happens with the character /. It denotes command options in the OS/2 program, but to REXX, it is the division operator:

```
w  = "anything"
dir/w
```

This would also result in a syntax error, because you can only divide with numbers. The correct way would be:

```
delete "*.bak"
dir"/w"
```

or

```
"delete *.bak"
"dir/w"
```

If you want to use a variable in the command string (see Figure 5-4 on page 5-4), leave the variable outside the quotes. For example:

```
extension = "BAK"
"delete *."||extension

option = "/w"
"dir"||option
```

To Summarize

The basic points of using commands are:

- REXX always examines each clause for REXX syntax.
- If REXX finds no valid instruction or label and the clause is not empty, then it evaluates the clause as an expression.
- The result of the expression, a string, is then passed on to the environment from which REXX was originally called—the default environment.

ADDRESS Instruction

A more explicit way to send a command to the environment is by using the ADDRESS instruction, ADDRESS environment expression, where:

environment is the destination of the string. To address the OS/2 program, use the symbol CMD.

expression is evaluated by REXX as a string to be passed to the environment.

The ADDRESS instruction works like this:

```
address CMD "dir"      /* pass the literal string   */
                       /* "dir" to the OS/2 program */

cmdstr = 'dir *.txt'   /* assign a string           */
                       /* to a variable             */

address CMD cmdstr     /* REXX passes the string          */
                       /* "dir *.txt" to the OS/2 program */
```

When you use the ADDRESS instruction this way, REXX first evaluates the string expression that follows the CMD. Then, it sends the resulting string to the OS/2 program.

The ADDRESS instruction in this example works exactly the same way as in the methods already described. If you want to send a literal string to the OS/2 program, you must enclose it in quotes. Otherwise, REXX evaluates the terms and operators and passes the resulting string to the OS/2 program.

The ADDRESS instruction lets a single REXX program issue commands to two or more environments. This can be used when using REXX as a macro language.

Test Yourself

Figure 5-7 shows a program that passes the DIR command and SORT filter to the OS/2 program.

```
/*                  */
say 'Type fragment'
pull frag
if frag = '' then
   say 'You asked for it!'
'DIR' frag ||'*.*  | SORT'
```

Figure 5-7. FILFRAG.CMD

1. What does each line of the program do?
2. What happens if the user types who at the prompt message?
3. What happens if the user presses the Enter key without typing anything?
4. What happens if no files fit the pattern?

Answers:

1. Here is what the program does:
   - The first line is a comment.
   - The second line displays the prompt message, `Type fragment:`, on the screen.
   - The third line reads the user's entry and stores it in the variable `frag`.
   - The fourth line tests if `frag` is empty, as in HELLO.CMD, see Figure 2-1 on page 2-1.
   - If `frag` is empty, then the fifth line displays the message, `You asked for it!`, on the screen.
   - The sixth line is evaluated as a string expression and concatenates the contents of `frag` into a DIR command with a wildcard file-pattern. The directory is sorted and displayed on screen.
2. The OS/2 program displays a directory for the command

   `DIR WHO*.* ¦ SORT`
3. The program displays the message `You asked for it!` and then a listing of all files in the current directory, as if the command `DIR *.* ¦ SORT` had been typed.
4. The OS/2 program displays the message, `The system cannot find the file specified.`, on the screen.

The following is displayed on screen when you run the program.

```
[C:\]filfrag
Type fragment
?
who
<sorted directory display>
[C:\]
```

The following is displayed on the screen if you make no entry and press the Enter key.

```
[C:\]filfrag
Type fragment
?

You asked for it!
< full directory listing >
[C:\]
```

The following is displayed on the screen if no files in the current directory match the given pattern.

```
[C:\]filfrag
Type fragment
?
who

SYS0002 The system cannot find the file specified.
```

From the OS/2 Program to REXX

Information between REXX and OS/2 programs is passed both ways; this is an essential part of using REXX with other environments. If a command fails to operate the way you intended, such as if you try to copy a file that does not exist, you get a message that tells you that file cannot be found and you can respond accordingly.

The same thing happens when a REXX program tries to copy a nonexistent file, except that for REXX to respond, it must be apparent that a system error has occurred. Otherwise, the program continues running, unaware that neither the source file nor the copy of it exists.

A system error alone does not stop the running of a REXX program. Without some provision to stop the program, called a *trap*, REXX continues running. You may have to press the Control (Ctrl)+Break keys to stop processing.

With each command it processes, the OS/2 program produces a number called a *return code*. When a REXX program is running, this return code is automatically assigned to a special built-in REXX variable named RC.

If the command was processed with no problems, the return code is nearly always 0. If something goes wrong, the return code issued is another nonzero number, depending on the command itself and the error encountered.

In the previous example, the *file not found* error occurred at the end of the program, so there was no great problem. But a system error that occurs in the middle of a program can indeed cause trouble.

Reading Return Codes

Figure 5-8 shows a program that can read a return code.

```
/* RC report */
"TYPE nosuch.fil"
say "the return code is" RC
```

Figure 5-8. GETRC.CMD

Figure 5-9 shows a program that displays a response to the return code.

```
/* Simple if/then error-handler. */
say "Type a file name:"
pull filename
"TYPE" filename
if RC \= 0
then say "Could not find" filename
```

Figure 5-9. REPORTRC.CMD

This program tells you only that the OS/2 program could not copy a nonexistent file. This program does not do much more than the OS/2 program does, but you have the basic idea of how to capture a return code.

Programs need to respond more effectively to whatever situation occurs. This applies not only to OS/2 errors, but also to the choices of a program user and even the data that the program processes. For that kind of response, REXX has instructions, such as IF, that control the processing of the program itself, sometimes called its *flow*. For more information about program flow, refer to Chapter 6, "Program Control."

More and Better Traps

When you have completed "Basics" in this guide, you may want to explore other methods of controlling system errors, particularly the instructions:

- CALL ON ERROR
- CALL ON FAILURE
- SIGNAL ON ERROR
- SIGNAL ON FAILURE.

These instructions are discussed in "Advanced Topics" in this chapter (see "Trapping Command Errors" on page 5-12) and "Instructions" in the *REXX Reference*.

Summary

This completes "Basics" in this chapter. You have learned how to:

- Pass commands from REXX programs to the OS/2 program:
  - Using expressions
  - Using the ADDRESS instruction.
- Read the RC variable that the OS/2 program returns to REXX.

"Advanced Topics" in this chapter discusses:

- Using REXX as a substitute for batch files
- Using REXX as a macro language
- Using SIGNAL and CALL to trap errors and failures.

To continue with "Basics," go to page 6-1.

Advanced Topics

In this chapter:

Advanced Topics

- REXX and Batch Files
- Subcommand processing
- Trapping command errors.

REXX and Batch Files

You can use a REXX program anywhere you now use OS/2 batch files. Figure 5-10 shows an example of an OS/2 batch file that processes user input to display a help message.

```
@echo off
: SCCSID = @(#)help.cmd 1.1 87/11/16
if %1.==. goto msg
if %1 == on goto yes
if %1 == off goto no
if %1 == ON goto yes
if %1 == OFF goto no
if %1 == On goto yes
if %1 == oN goto yes
if %1 == OFf goto no
if %1 == OfF goto no
if %1 == Off goto no
if %1 == oFF goto no
if %1 == oFf goto no
if %1 == ofF goto no
helpmsg %1
goto exit
:msg
helpmsg
goto exit
:yes
prompt $ifm$p"
goto exit
:no
cls
prompt
:exit
```

Figure 5-10. HELP.CMD (OS/2-batch version)

Figure 5-11 shows an example of an equivalent program in REXX.

```
/* HELP.CMD - Get help for a system message */
arg action .
select
  when action=''    then     'helpmsg'
  when action='ON'  then     'prompt $i$p$p'
  when action='OFF' then do
                           'cls'
                           'prompt'
                           end
  otherwise 'helpmsg' action
  end
exit
```

Figure 5-11. HELP.CMD (REXX version)

Subcommand Processing

REXX programs can issue commands or subcommands to programs other than the OS/2 program. The specifics about other application environments are beyond the scope of this book. However:

- If your application permits access to the OS/2 system prompt, you can call REXX programs that way.
- To use REXX as a macro or scripting language in applications such as word processors, the application must be *registered* with the language processor. This is done by the producer of the program.
- If you have programs of your own that you want to register with the language processor, refer to "Applications Programming Interfaces" in the *REXX Reference*.

Trapping Command Errors

The most efficient way to detect errors from commands is by creating *condition traps*, using the SIGNAL ON and CALL ON instructions, with either the ERROR or the FAILURE condition. When used in a program, these instructions *enable* (switch on) a detector in REXX that tests the result of every command. Then, if a command signals an error, REXX stops normal program processing, searches the program for the appropriate label (`ERROR:` or `FAILURE:`) or a label that you created, and resumes processing there.

The SIGNAL ON and CALL ON instructions also tell REXX to store the line number (in the REXX program) of the command instruction that triggered the condition. That line number is assigned to the special variable SIGL. Your program can get even more information about what caused the command error through the built-in function CONDITION().

Using the SIGNAL and CALL instructions to handle errors has several advantages. Programs:

- Are easier to read, because you can confine error-trapping to a single, common routine

- Are more flexible, because they can respond to errors by clause (SIGL), by return code (RC), or by other information (CONDITION())
- Can catch problems and react to them before the environment issues an error message
- Are easier to correct, because you can turn the traps on and off (SIGNAL OFF and CALL OFF).

For other conditions that may be detected using SIGNAL ON and CALL ON, see "Condition Traps" on page 7-15.

Instructions and Conditions

The instructions to set a trap for errors are:

```
SIGNAL ON condition [NAME trapname]
CALL   ON condition [NAME trapname]
```

SIGNAL ON Initiates an exit subroutine that ends the program

CALL ON Initiates a return subroutine that returns processing to the clause immediately following the CALL ON instruction. Use this instruction to *recover* from a command error or failure.

The two command conditions that can be trapped are:

ERROR Detects *any* nonzero error code issued by the default environment as the result of a REXX command

FAILURE Detects a severe error preventing the system from processing the command.

A failure, in this sense, is a particular category of error. If you use SIGNAL ON or CALL ON to set a trap only for ERROR conditions, then it traps failures as well as other errors. If you also specify a FAILURE condition, then the ERROR trap ignores failures.

As an option, you can also use the NAME keyword to specify a particular subroutine, trapname, to be run. When an ERROR or FAILURE condition occurs and:

- If a trap is specified by name, REXX jumps to the clause following the appropriate label (the trapname followed by colon).
- If you do not specify a trapname in the SIGNAL ON or CALL ON instruction, REXX searches for a label matching the appropriate condition. It looks for the label ERROR: or FAILURE:).

For more information about other conditions that can be trapped, see "Condition Traps" on page 7-15. For more information about how labels are used to call subroutines, refer to Chapter 7, "Program Structure."

Disabling Traps

To turn off a trap for any part of a program, use the same instruction with the OFF keyword, such as:

```
SIGNAL OFF ERROR
SIGNAL OFF FAILURE
CALL OFF ERROR
CALL OFF FAILURE
```

Using SIGNAL ON ERROR

Figure 5-12 shows an example of how a program might use SIGNAL ON to trap a command error in a program that copies a file. In this example, an error occurs because a nonexistent file name is stored in the variable FILE1. Processing jumps to the clause following the label error:.

```
/* example of error trap */
signal on error                 /* Set the trap */
     .
     .
     .
"COPY"  file1 file2             /* When an error occurs... */
     .
     .
     .
error:                           /* ...REXX jumps to here  */
say "Error" rc "at line" sigl
say "Program can not continue."
exit                             /* and ends the program.  */
```

Figure 5-12. SIGNAL ON Trap

Using CALL ON ERROR

If there were a way to recover, such as by typing another file name, you could use CALL ON as shown in Figure 5-13 to recover and resume processing.

```
/* example of error recovery */
call on error
     .
     .
     .
"COPY"  file1 file2
say "Using" file2 <-------------------------------
     .                                           |
     .                                           |
     .                                           |
error:                                           |
say "Can not find" file1                         |
say "Type Y to continue anyway."                 |
pull ans                                         |
if ans = "Y" then                                |
   do                                            |
      /* create dummy file */                    |
           .                                     |
           .                                     |
           .                                     |
      file2 = "dummy.fil"                        |
      RETURN ------------------------------------
      end
else exit
```

Figure 5-13. CALL ON Trap

A Common Error-handling Routine

Figure 5-14 shows an example of a simple error trap that can be used in many programs.

```
/* Here is a sample "main program"          */
signal on error       /* enable error handling          */
'ersae myfiles.*'     /* mis-typed 'erase' instruction  */
exit

/* And here is a fairly generic error-handler for this */
/* program (and many others...)                        */
error:
  say 'error' rc 'in system call.'
  say
  say 'line number =' sigl
  say 'instruction = '  || sourceline(sigl)
  exit
```

Figure 5-14. SIGERR.CMD

Chapter 6. Program Control

So far, the sample programs have been fairly straightforward lists of clauses. Various instructions have been used to input, store, display, and manipulate information.

In this chapter, another class of instructions, *keywords* that manipulate the program itself, are discussed. One of these, IF... THEN... ELSE, was already used to choose between two possible directions a program might take.

Basics

In this chapter:

Basics

- Changing the flow of a program
- Repetitive tasks
- Conditional loops
- Using counters to exit loops
- Exiting a program.

Changing the Flow of a Program

A program can be:

- A single list of instructions
- A number of short lists connected by instructions that determine which list to process and how many times to process it.

Instructions that direct the processing of the program are called *control instructions.* The maneuvers they perform include:

Branching Selecting one of several lists of instructions to process. The branching instructions are IF and SELECT.

Looping Repeating a list of instructions, either for a specified number of times or as long as some condition is satisfied. The DO instruction (when used with keywords like UNTIL and WHILE) does the looping in REXX.

Exiting A program that is a single list ends when it reaches the last instruction. To explicitly end a program, use the instructions EXIT and RETURN.

Grouping Instructions

What most control instructions have in common is that they often use groups of clauses that act as a single clause. The simplest way to group clauses is with the keyword DO. For example:

```
DO
   clause1
   clause2
   clause3
   ...
END
```

If the keyword DO is in a clause by itself, the list of clauses that follows (up to the END keyword) is processed once (no loop is implied). This form of the DO instruction and the END keyword associated with it tell REXX to treat the enclosed instructions as a single instruction.

The enclosed instructions may be indented to the right. The indentation does not affect how REXX processes the list. It does, however, make the program easier for people to read. It shows that these instructions belong together.

Testing Conditions

In Chapter 4, "Expressions," a class of expressions called *comparisons*, which test whether a given condition is true or false, was introduced. In a comparison, two terms are joined by operators such as = (equal to), > (greater than), or < (less than) in order to pose a test of the terms. The expression itself evaluates as, 1 if the comparison is *true* or 0 if the comparison is *false*. For example:

```
/* some comparisons */
say 5 = 5                  /* displays '1' - true    */
say 5 < 4                  /* displays '0' - false   */
say 5 = 4                  /* displays '0' - false   */
say 5 > 4                  /* displays '1' - true    */

reply = "YES"              /* assigns the string "YES"
                             to the variable REPLY   */

say reply                  /* displays "YES"         */
say reply = "MAYBE"        /* displays '0' - false   */
say reply = "YES"          /* displays '1' - true    */
```

For more complex ways to combine terms and operators to form comparisons, see "Comparisons" on page 4-13.

Simple Branching

To tell REXX how to make a decision about a single instruction, use:

```
IF expression
THEN instruction
```

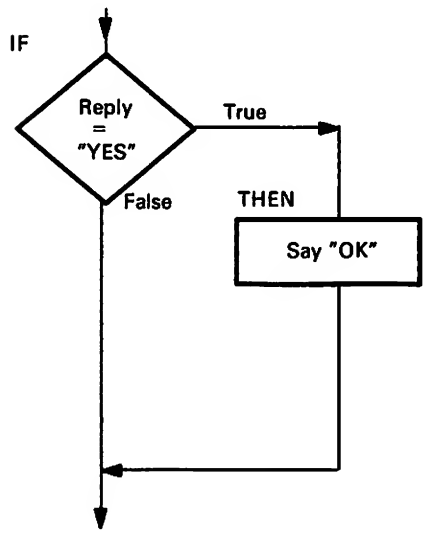
REXX processes `instruction` only if `expression` is true, see Figure 6-1.

```
/* Asking confirmation */
say "Type YES to continue"
pull reply
if reply = "YES" then say "OK!"
/* program continues from here...*/
```

Figure 6-1. CONFIRM.CMD

The instruction `say "OK!"` is processed only if the variable `reply` has the value `YES`.

The IF instruction introduces a new *branch* of instructions to process when the IF expression is true. Programmers often visualize the action of a decision-making instruction by using a diagram like this, called a *flowchart*.



The decision-making expression is represented by a diamond. If the expression (here REPLY="YES") evaluates as *true*, then the program branches, or takes a detour, through one additional instruction before resuming on the next line.

Using DO...END for Multiple Clauses

To put a list of instructions after the THEN, use the DO instruction and the END keyword. That turns the whole group into a single instruction. For example:

```
IF expression THEN
DO
  instruction1
  instruction2
  instruction3
  <...and so on>
END
```

With the DO and END keywords bracketing the list, REXX knows to treat the listed instructions as a unit to:

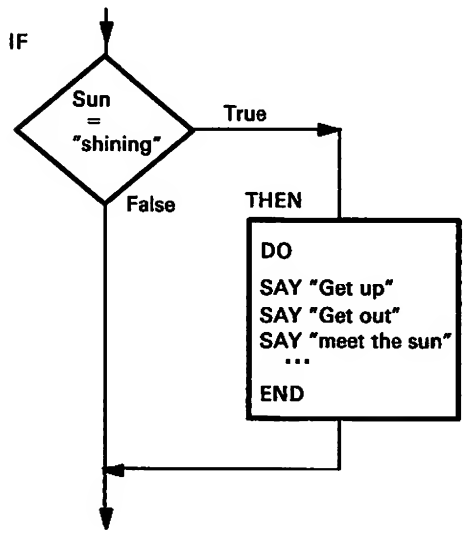
- Process all of them if expression is true
- Ignore them all if expression is false.

Figure 6-2 shows an example using DO and END.

```
/* Wake-up call */
if sun = "shining"
then
   do
     say "Get up!"
     say "Get out!"
     say "Meet the sun half way!"
   end
```

Figure 6-2. WAKEUP.CMD

The flowchart diagram would look like this.



In the previous example, if `sun = "shining"` evaluates as 1 (true), then all three SAY instructions are processed. But if `sun = "shining"` evaluates as 0 (false), then none of the SAY instructions are processed.

The THEN and DO keywords are each on a separate line. This is optional. You could also write the program as:

```
/* ...this way... */
if sun = "shining" then
do
  say "Get up!"
  (etc.)
  end
```

```
 /* ...or this way */
If sun = "shining" then do
   say "Get up!"
   (etc.)
   end
```

Test Yourself

Refer to Figure 6-2 on page 6-4. What would happen if you left out DO and END keywords?

Answer: Without DO and END to mark the list as a unit, REXX assumes that only the first instruction following THEN is processed on condition.

The instruction, `say "Get up!"`, would be processed only if the comparison expression, `sun = "shining"`, is true. The rest of the instructions in the list would always be processed, regardless of the true\or-false condition of `expression`.

Two Paths: ELSE

Used alone, IF...THEN adds a branch of instructions to process when the controlling expression is true. You can also add a second branch of instructions to process when the expression is false. The keyword ELSE introduces this alternate list. For example:

```
IF expression
THEN instruction1
ELSE instruction2
```

When IF is used this way, REXX processes only one of these instructions, not the other. It will process:

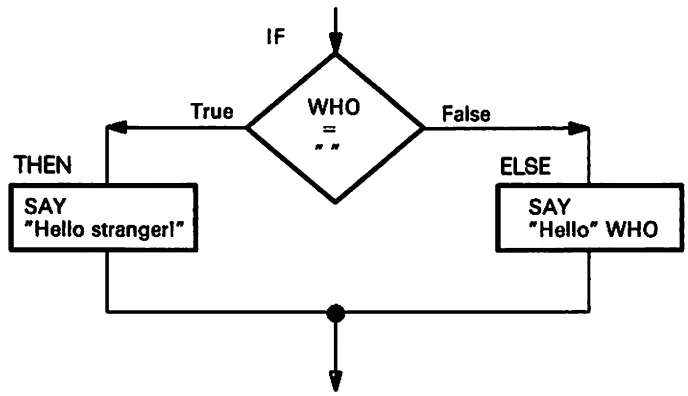
- `Instruction1` only if `expression` is true
- `Instruction2` only if `expression` is false.

IF, THEN, and ELSE were used to control the processing of the first program in this book, HELLO.CMD, as shown in Figure 6-3.

```
/* A conversation */
say "Hello! What is your name?"
pull who
if who = "" then say "Hello stranger"
else say "Hello" who
```

Figure 6-3. HELLO.CMD

The flowchart diagram would look like this.



The same idea, with a command, is used to create a more practical program. Figure 6-4 shows a program that takes a file name and creates a backup copy. You may want to compare the program shown in Figure 6-4 to the REPORTRC.CMD program shown in Figure 5-9 on page 5-10.

```
/* backup a REXX program */
arg fname"."ext                  /* This is a technique called    */
                                 /* "parsing a literal pattern".  */
                                 /* Use it just as you see it     */
                                 /* here for now; more about it   */
                                 /* on page 8-10.                  */

if fname = "" then do            /* If no file name typed, then   */
   Say "Type a file name:"       /* prompt user to type a name    */
   pull fname"."ext
   end

if ext = "" then ext = "CMD"     /* IF no extension given, then   */
                                 /* give it the extension '.CMD'  */

"dir" fname"."ext                /* displays directory entry for  */
                                 /* the input file name, thereby  */
                                 /* making sure it exists....     */

if rc <> 0 then do               /* IF no such file exists, then  */
   say "No backup performed."    /* EXIT the program. (More about */
   say "Program ended."          /* the EXIT instruction on       */
   exit                          /* on page 6-23.)                 */
   end

else do                                 /* ELSE, copy the file to  */
   say "Backing up" fname"."ext         /* one with the extension  */
   "copy" fname"."ext  fname".BKP"      /* '.BKP'; then confirm    */
   "dir" fname".BKP"                    /* it with a "DIR" command */
   say "Program ended"
   exit
   end
```

Figure 6-4. BACKITUP.CMD

The ELSE clause must also use DO and END to bracket a list of instructions.

Note: REXX features in this program that have not been introduced yet are:

- The EXIT instruction, which tells REXX explicitly to end the program.
- The period in quotes in the ARG and PULL instructions called a *literal parsing pattern*, which tells REXX to remove that quoted string (if it occurs) in the user's file name entry. What is left is broken into two parts, which in turn are assigned to the variable FNAME and EXT. This is the only example of a literal pattern used in "Basics". For more information, refer to "Parsing with Patterns" on page 8-10.

The SELECT Instruction

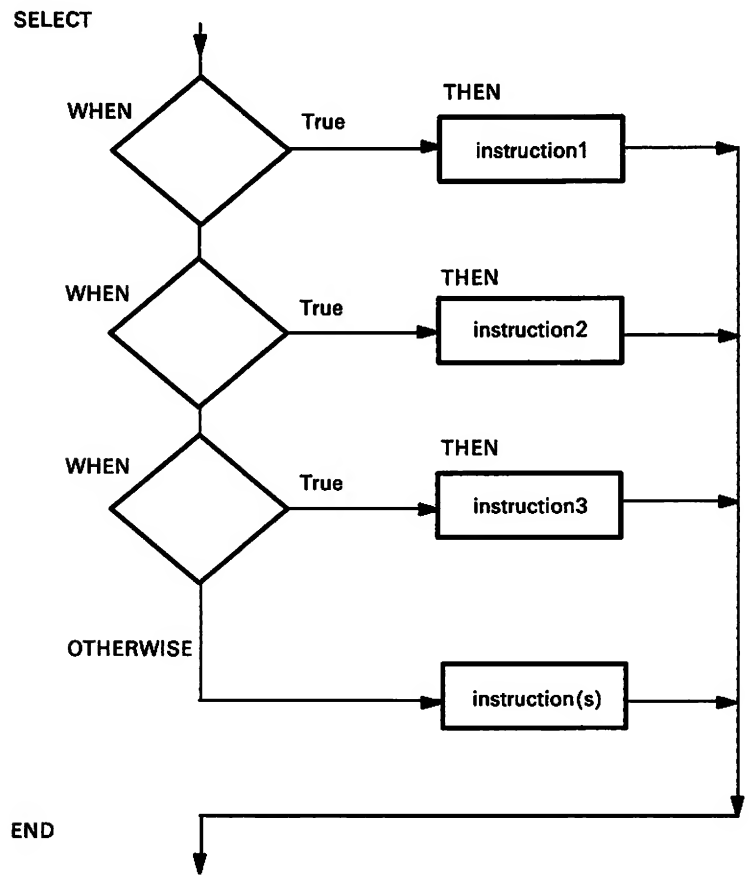
You are not limited to two choices. You can use the SELECT instruction to have a REXX program select one of any number of branches. For example:

```
SELECT
   WHEN expression1 THEN instruction1
   WHEN expression2 THEN instruction2
   WHEN expression3 THEN instruction3
   ...

    OTHERWISE
       instruction
       instruction
       instruction
       ...

END
```

- If expression1 is true, instruction1 is processed. After this, processing continues with the instruction following the END.
- If expression1 is false, then expression2 is tested. If it is true, then instruction2 is processed and processing continues with the instruction following the END.
- If expression1, expression2, and so on, are all *false*, then processing continues with the instruction following the OTHERWISE.

OTHERWISE is essentially the SELECT-equivalent of ELSE. If there is any possibility that all the WHEN expressions could be false, there must be an OTHERWISE clause.

The following is how SELECT is diagramed in a flowchart.



To process a list of instructions following the THEN keyword, use:

```
DO
   instruction1
   instruction2
   instruction3

:
END
```

A DO; ... END group is not required after the OTHERWISE keyword.

Multiple Choice

Figure 6-5 shows a short program that uses SELECT.

```
/* displays date/time information */

arg request      /* get argument; covert to uppercase */

select
   when request = "DATE"  then say date()
   when request = "TIME"  then say time()
   when request = "DAY"   then say date(w)
   when request = "MONTH" then say date(m)
   when request = "SOFAR" then do
                                 say time(h) "hours"
                                 say time(m) "minutes"
                                 say time(s) "seconds"
                               end

   /* if no valid argument given, display help information */

   otherwise say "Valid arguments are:"
             say " date   -  calendar date"
             say " time   -  military time"
             say " day    -  day of the week"
             say " month  -  month"
             say " sofar  -  hrs/min/sec since midnight"
end
```

Figure 6-5. Q.CMD

Remember that SELECT must have a corresponding END. As with DO, it makes your program easier for people to read if you indent everything between the SELECT and the END three spaces to the right. SELECT is a specialized form of the IF instruction.

Test Yourself

Write a program that asks the user to type two words on the same line and computes if:

- The values of the words are the same (or numerically equal).
- The value of the first word is higher.
- The value of the second word is higher.

The comparison must ignore differences in case. For example, A should evaluate as equal to a.

Answers: Figure 6-6 shows one possible program.

```
/* This program requests the user to supply two     */
/* words and says which is higher.                  */

say "Type two words"
pull word1 word2 .
select
   when word1 = word2 then
      say "The words are the same",
               "or numerically equal"
   when word1 > word2 then
      say "The first word is higher"
   otherwise
      say "The second word is higher"
   end
```

Figure 6-6. COMPARE1.CMD

Figure 6-7 shows an alternative program using IF.

```
/* This program requests the user to supply two     */
/* words and says which is higher.                  */

say "Type two words"
pull word1 word2 .
if word1 = word2
then say "The words are the same",
           "or numerically equal"
else do
   if word1 > word2
   then say "The first word is higher"
   else say "The second word is higher"
end
```

Figure 6-7. COMPARE2.CMD

Some people may consider the first solution better because it is slightly easier to understand.

For other considerations about when to use IF and SELECT, see "Nested IF and SELECT" on page 6-26.

Repetitive Tasks

Computers excel at repetitive tasks. An essential part of any computer language is a *loop* instruction, which is a way to make a program repeat a list of instructions:

- A specific number of times
- As long as some condition is true
- Until some condition is satisfied
- Forever (until the user wants to stop).

To repeat a loop a number of times, use:

```
DO exprr
   instruction1
   instruction2
   instruction3

:
END
```

where:

exprr (the **expr**ession for **r**epetitor) gives a whole number, which is the number of times the loop is processed.

To make your program easier for people to read, you should indent the instructions between the DO and the END three spaces to the right.

Figure 6-8 is an example of a repetitive loop that prints three documents five times.

```
/* To print documents for a meeting:  for each person,  */
/* the agenda, minutes and accounts are printed once    */

do 5
   "PRINT AGENDA.DOC"
   "PRINT MINUTES.DOC"
   "PRINT ACCOUNTS.DOC"
end
```

Figure 6-8. HANDOUTS.CMD

Consider how you would write a program that would *ask* for the number of copies needed and one that would ask for the names of the documents.

Figure 6-9 is an example of a repetitive loop that processes the instruction between the DO and the END, height times.

```
/* The user is asked to specify the height of a        */
/* rectangle (within certain limits).  The rectangle   */
/* is then displayed on the screen.                    */

say "Type the height of the rectangle",
    " (a whole number between 3 and 15)."
pull height
select
   when  \datatype(height,WHOLE) then say "Rubbish!"
   when  height < 3 then say "Too small!"
   when  height > 15 then say "Too big!"
   otherwise

                                /* draw rectangle      */
   do height
      say copies("*",2*height)
   end

   say "What a pretty box!"
end
```

Figure 6-9. RECTANGL.CMD

Conditional Loops

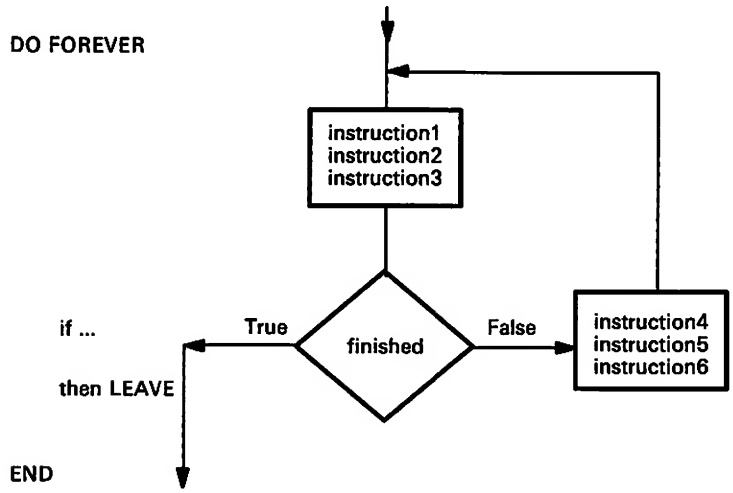
A conditional expression is tested to determine how many times the loop is processed. Conditional loops continue running as long as some condition is satisfied. The three main ways to do this, depending on when the test takes place are:

- DO FOREVER (with LEAVE)
- DO WHILE (a condition is true)
- DO UNTIL (a condition is true).

Note: When you are experimenting with conditional loops, you may run into a situation where your program does not stop. This is called an *endless loop*, which means that the condition that controls the loop is never false. If this happens, you can stop the program by pressing the Control (Ctrl)+Break keys.

DO FOREVER with the LEAVE Instruction

The simplest way to create a conditional loop is to use the instructions DO FOREVER and LEAVE. The LEAVE instruction causes processing to continue with the instruction following the END keyword.



The mini-calculator program, ADD2NUM.CMD, shown in Figure 3-1 on page 3-1, adds only two numbers. Figure 6-10 shows a program that continues running as long as you type a number. If you do not type a number, the LEAVE instruction is processed and processing continues with the SAY instruction after the END of the loop.

```
/* This program adds up the numbers that the user is    */
/* invited to type.  When the user types something      */
/* that is not a number, a message is displayed and     */
/* the program ends.                                    */
total = 0
do forever
   say "Type a number"
   pull entry
   if \datatype(entry,n)  /*if the entry is not a valid number */
   then leave                    /* leave the loop */
   total = total + entry
   say "Total = " total
end
say "'"entry"' is not a number.  Returning to OS/2."
```

Figure 6-10. SUM.CMD

The other two forms of DO loops, where the conditional test (the `if`) is built into the control instruction, are:

- DO WHILE
- DO UNTIL.

DO WHILE Instruction

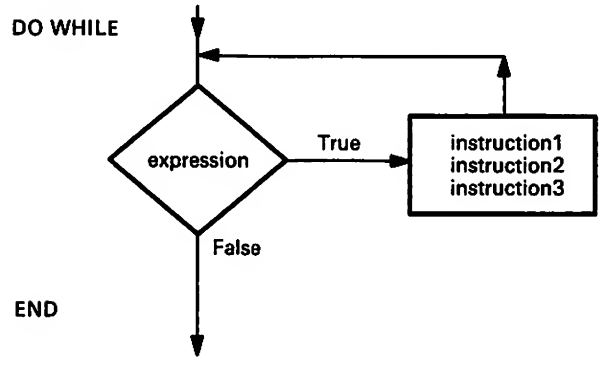
To create a loop that repeats the list of instructions as long as a given condition is true, use the DO WHILE instruction. For example:

```
DO WHILE exprw
   instruction1
   instruction2
   instruction3
END
```

where:

exprw (**expr**ession for **w**hile) is an expression that, when evaluated, must give a result of 0 or 1.

The following is a flowchart of a DO WHILE loop. The condition is tested at the top of the loop, before the instruction list is processed. This means that if the given condition is false at the start, the list of instructions are not be processed at all. Compare this instruction with the flowchart showing the DO UNTIL instruction on page 6-15.



Two *fragments* that produce the same results are:

```
DO WHILE \ finished
   instruction1
   instruction2
   instruction3
END
```

or

```
DO FOREVER
   if finished then LEAVE
   instruction1
   instruction2
   instruction3
END
```

You could use DO WHILE to prompt users for data only if they forget to type an *argument* at the command prompt. For example:

```
/* get the argument */
arg filename
do while filename = ""    /* if no argument given, then do... */
  say "Type a file name (or a * to quit):"
  pull filename
  if filename = "*" then exit
  end
⋮
```

In the previous example, if the user types a file name as a command argument, then the instructions within the DO WHILE loop are ignored. If no argument has been given, the loop is processed as long as the variable `filename` holds an empty string.

DO UNTIL Instruction

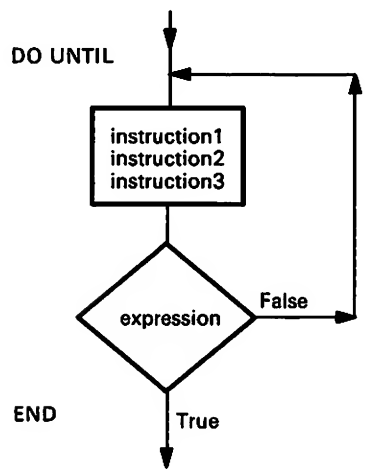
To repeat one or more instructions *until* a given condition is true, you can create a loop with the test at the bottom. For example:

```
DO UNTIL expru
   instruction1
   instruction2
   instruction3

⋮
END
```

where:

expru (**expr**ession for until) is an expression that, when evaluated, must give a result of 0 or 1.

Putting the test at the bottom of the loop means that the enclosed instruction list is always processed *at least once*, even if the given condition is false at the start. Compare the following flowchart with the previous one for the DO WHILE instruction on page 6-14.



Two fragments that produce the same results are:

```
DO UNTIL finished
   instruction1
   instruction2
   instruction3
END
```

or

```
DO FOREVER
   instruction1
   instruction2
   instruction3
   if finished then LEAVE
END
```

The DO UNTIL instruction also provides a convenient way to check input. This loop uses the DATATYPE() function to ensure that the user types only a number:

```
/* numbers only */
do until datatype(entry,num)
  say "type a number"
  say "(or press the Enter key alone to quit):"
  pull entry
  if entry = "" then exit
  end
```

This loop is always processed at least once, even if the variable entry is already a number. It continues until the user either types a number or enters an empty string by pressing the Enter key.

By typing an asterisk in the first example or a null string in the second, the user has a way out of the loop. It is important to include these *escape clauses*. Endless loops are frustrating to the program user.

To Summarize

In the three kinds of conditional loops, the decision is made:

- *Before* processing starts. The following example of this program *fills* bath by repeatedly adding the value of bucket to it. If bath is already full (equal to or greater than the value of full), the body of the loop is not processed and nothing is added to bath.

  ```
  DO WHILE bath < full
     bath = bath + bucket
  end
  ```

- *After* the first pass through the loop and again after every subsequent pass. An example is requesting valid data from a user.

  ```
  DO UNTIL datatype(input,NUMBER)
     say "Type a number"
     pull input
  end
  ```

- *During* each pass. For example, the decision to leave may depend on information obtained during the loop.

```
DO FOREVER
   say "Type an item of data.  When there is",
       " no more data, type QUIT"
   pull answer
   if answer = "QUIT" then leave
      ...   /* process the data */
end
```

Be careful about the condition for repeating the loop. For DO WHILE, the condition must be *true*; for DO UNTIL, the condition must be *false*.

Test Yourself

1. What type of DO instruction would you use to code the following sequence?

   Is job done?
   instruction1
   instruction2
   instruction3
   Is job done?
   instruction1
   instruction2
   instruction3
   Is job done?

   ⋮

   Is job done?

2. What type of DO instruction would you use to code the following sequence?

   instruction1
   instruction2
   instruction3
   Is job done?
   instruction1
   instruction2
   instruction3
   Is job done?

   ⋮

   Is job done?

3. "Thirty days hath September, April, June, and November; all the rest have thirty-one, excepting February alone ..."

   Write a program that asks the user to specify the month as a number between 1 and 12 and gives the number of days in the month in response. For month 2, the response can be 28 or 29.

Answers:

1. DO WHILE job \= done (The first operation is to test "Is job done?").
2. DO UNTIL job = done (The first operation is to process the list of instructions.)
3. Figure 6-11 shows a program that displays the number of days in the month.

```
/* This program requests the user to type a whole   */
/* number from 1 through 12 and displays the        */
/* number of days in that month.                    */

/*--------------------------------------------------*/
/* Get input from user                              */
/*--------------------------------------------------*/
do until datatype(month,WHOLE),
      & month >= 1 & month <= 12
   say "Type the month as a number from 1 through 12"
   pull month
end
/*--------------------------------------------------*/
/* Compute days in month                            */
/*--------------------------------------------------*/
select
   when month =  9 then days = 30
   when month =  4 then days = 30
   when month =  6 then days = 30
   when month = 11 then days = 30
   when month =  2 then days = "28 or 29"
   otherwise
   days = 31
end

say "There are" days "days in Month" month
```

Figure 6-11. CALENDAR.CMD

Using IF, SELECT, and DO

Figure 6-12 shows a program that combines three different control instructions. It asks the user to provide a person's age and sex and, in reply, it displays a person's status.

Note:

A person under the age of 5 is a BABY.
A person aged 5 through 12 is a BOY or a GIRL.
A person aged 13 through 19 is a TEENAGER.
A person over the age of 19 a MAN or a WOMAN.

The program uses DO UNTIL to make certain that the proper input has been typed. It then uses SELECT to choose one of four age groups and IF, as needed, to determine the sex. Try the program.

```
/*---------------------------------------------------*/
/* Get input from user                               */
/*---------------------------------------------------*/
do until datatype(age,NUMBER) & age >= 0
   say "What is the person's age?"
   pull age
end

do until sex = "M" ¦ sex = "F"
   say "What is the person's sex (M or F)?"
   pull sex
end
/*---------------------------------------------------*/
/* COMPUTE STATUS                                    */
/*                                                   */
/* Input:                                            */
/* AGE     Assumed to be 0 or a positive number.     */
/* SEX     "M" is taken to be male;                  */
/*         anything else is taken to be female.      */
/*                                                   */
/* Result:                                           */
/* STATUS  Possible values: BABY, BOY, GIRL, TEENAGER*/
/*         MAN, WOMAN.                               */
/*---------------------------------------------------*/
Select
   when age < 5 then status = "BABY"
   when age < 13 then do
      if sex = "M"
      then status = "BOY"
      else status = "GIRL"
      end
   when age < 20 then status = "TEENAGER"
   otherwise
      if sex = "M"
      then status = "MAN"
      else status = "WOMAN"
end

say "This person should be counted as a" status
```

Figure 6-12. CENSUS.CMD

Using Counters to Exit Loops

Number each pass through the loop in such a way that you can use that number as a variable in your program. For example:

```
DO name = expri
   instruction1
   instruction2
   instruction3

:
END
```

or

```
DO name = expri TO exprt
   instruction1
   instruction2
   instruction3

:
END
```

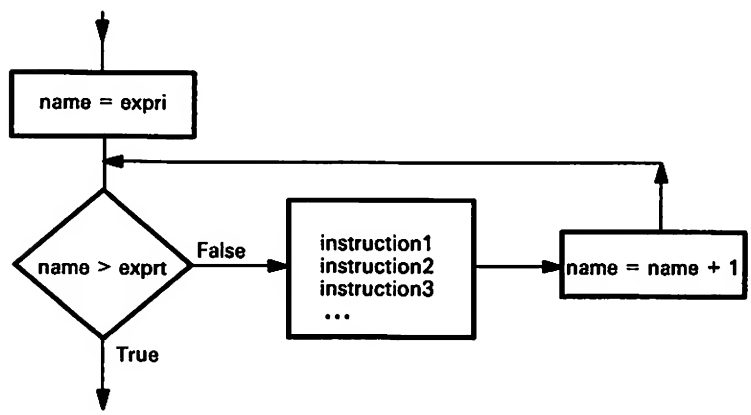
where:

name — is the control variable, sometimes called a *counter*. You can use it in the body of the loop. Its value is changed (in this example, increased by 1) each time through the loop.

expri — (the **expression** for the **initial** value) is the value you want the counter to have the first time through the loop.

exprt — (the **expression** for the **TO** value) is the value you want the counter to have the last time through the loop. That is, the loop ends if the next time through, it puts the counter above the *exprt* value.

The following flowchart shows how the counter is changed and how the decision to leave the loop is made.



You can use the counter to compute something different each time through the loop. Figure 6-13 shows a program in which the counter is called count, and it computes the width of each row of asterisks.

```
/* This program displays a triangle on the screen.      */
/* The user is asked to specify the height of the       */
/* triangle.                                            */

say "Type the height of the triangle",
    " (a whole number between 3 and 15)."
pull height
select
   when  \datatype(height,WHOLE) then say "Rubbish!"
   when  height < 3 then say "Too small!"
   when  height > 15 then say "Too big!"
   otherwise

                                /* draw triangle       */
   do count = 1 to height
      say copies("*",2*count - 1)
   end

   say "What an ugly triangle!"
end
```

Figure 6-13. TRIANGLE.CMD

After you have left the loop, you can still refer to the counter. It always exceeds the value of the TO expression (exprt).

Bigger Steps

So far, the counter has been incremented by 1 each time through the loop. This is the default. To specify some other value, write:

```
DO name = expri BY exprb [TO exprt]

:

:
END
```

where:

exprb (the **expression** for **BY**) is the number that is to be added to name at the bottom of the loop.

Different Steps

All of the expressions described for controlling loops (the TO and BY expressions and the counter) need not be positive whole numbers. You can use expressions that evaluate as decimal fractions and negative values as well. See the *REXX Reference* for examples.

Test Yourself

1. Refer to the flowchart on page 6-20 and predict what the program in Figure 6-14 will display.

```
/* Example: use of a counter */
do digit = 1 to 3
   say digit
end
say "Now you have reached" digit
```

Figure 6-14. MORE.CMD

2. What will the program in Figure 6-15 display?

```
/* Example: use of a counter */
do count = 10 by -2 to 6
   say count
end
say "Now you have reached" count
```

Figure 6-15. 2LESS.CMD

3. How many lines will the program in Figure 6-16 display?

```
/* Example: use of a counter */
do j = 10 to 8
   say "Hup! Hup! Hup!"
end
```

Figure 6-16. 3HUP.CMD

4. How many lines will the program in Figure 6-17 display?

```
/* Example: use of a counter */
do NOW = 1
   if NOW = 9 then exit
   say NOW
end
```

Figure 6-17. 4NOW.CMD

Answers:

1. The counter is updated at the bottom of the loop. The test for leaving is made after this. So the counter is beyond the limit value.

   1
   2
   3
   Now you have reached 4

2. If exprb is negative, count down:

   10
   8
   6
   Now you have reached 4

3. None (10 already exceeds 8).

4. Eight, (on the ninth pass, the EXIT instruction ends the program before the SAY instruction is reached).

Exiting a Program

Use EXIT [expression], to tell REXX to leave your program. If you started the program by typing its name at the OS/2 command prompt:

- EXIT takes you back to the OS/2 program.
- The result of expression must be a whole number, which is returned to (but not displayed by) the OS/2 program.

Figure 6-18 shows an example using EXIT.

```
/* Example: using EXIT with a return code */
say "Returning to the OS/2 program"
exit 22
```

Figure 6-18. FADE.CMD

The following is displayed on the screen, when you run this program.

```
[C:\]fade
Returning to OS/2
[C:\]
```

Summary

This completes "Basics" in this chapter. You have learned how to:

- Create branches in a program with IF and SELECT
- Group instructions with DO
- Create loops with DO FOREVER, DO UNTIL, and DO WHILE.

"Advanced Topics" in this chapter discusses complex controls, including:

- Nesting IF instructions
- Using the NOP instruction
- Using DO with LEAVE and ITERATE
- Nesting DO instructions.

To continue with "Basics," go to page 7-1.

Advanced Topics

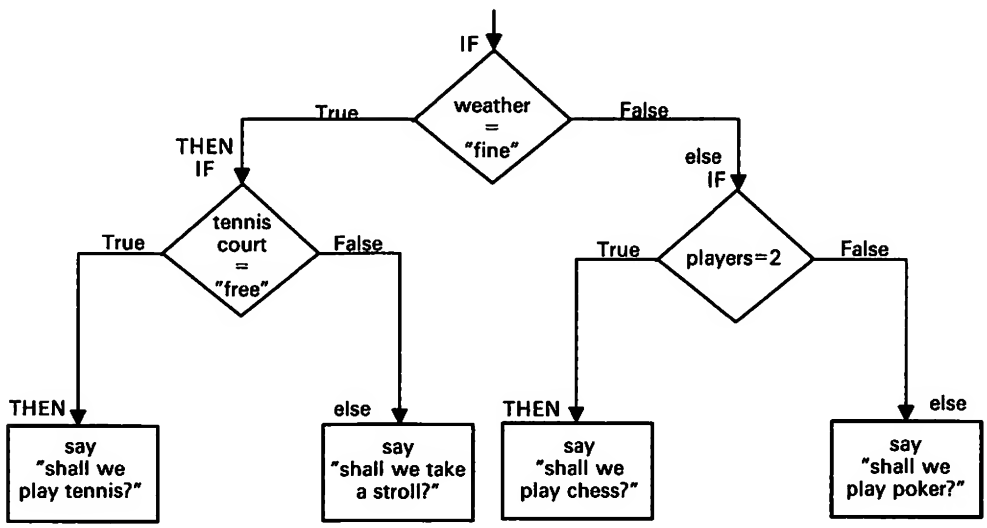
In this chapter:

Advanced Topics

- Nesting IF instructions
- The ITERATE instruction
- Compound DO instructions
- Nested loops.

Nesting IF Instructions

You can manage more complicated situations by using IF instructions in the lists controlled by other IFs. This flowchart shows two successive decisions that lead to one of four possible outcomes.



The best way to write this as a program is:

```
if weather = "fine"
then do
   if tenniscourt = "free"
   then say "Shall we play tennis?"
   else say "Shall we take a stroll?"
end
else do
   if players = 2
   then say "Shall we play chess?"
   else say "Shall we play poker?"
end
```

Indenting the secondary decisions to the right three spaces does not change how REXX processes the program. It makes it easier for someone reading the program to see the control structure.

Nested IF and SELECT

The previous example tests each condition and moves on to the next level. It does not consider whether the tennis court is free until it has determined that the weather is fine.

Compare this nested-IF example to one using the SELECT instruction. There still are four possible outcomes, but this time they are tested in parallel.

```
select
  when weather = "fine" & tenniscourt = "free"
     then say "Shall we play tennis?"
  when weather = "fine" & tenniscourt \= "free"
     then say "Shall we take a stroll?"
  when weather \= "fine" & players = 2
     then say "Shall we play chess?"
  otherwise say "Shall we play poker?"
end
```

The distinction to make here is not which version *works better.* Rather, it is which of the two programs is more *readable* to the user who corrects and improves them. The results of these two programs would be the same.

For this application, the nested-IF version shows more clearly how the decision whether to play tennis depends on the weather. The only priority of decision-making available to SELECT is the order given the WHEN keywords. Therefore use:

- SELECT when your program must make more or less parallel decisions, choosing one option to the exclusion of the rest.
- Nested IF when your program must make a series of decisions, each decision dependent on the ones that precede it.

Dangling ELSE

DO and END help REXX keep the ELSEs tied to the right IFs. Look at the following example. Avoid writing code like this, because it is too error-prone.

```
/* The dangling ELSE */

if weather = fine
then
    if tenniscourt = free
    then say "Shall we play tennis?"

  else say "Shall we take our raincoats?"

  /* REXX will take this ELSE to belong              */
  /* to the nearest preceding IF, but a person       */
  /* reading the program might easily assume that it */
  /* belonged to the first IF.                       */
```

Programs that have IFs within IFs should use DO ... END. The following example pairs THEN DO with END and THEN with ELSE.

```
if ...
   then do
      if ...
         then do
            ...
            ...
         end
         else do
            ...
            ...
         end
   end
   else ...
```

Remember, using indentation does not affect how the program is interpreted. It is simply a convention to make the program more readable. Readability is discussed in "Making Programs Easy to Read" on page 11-11.

Test Yourself

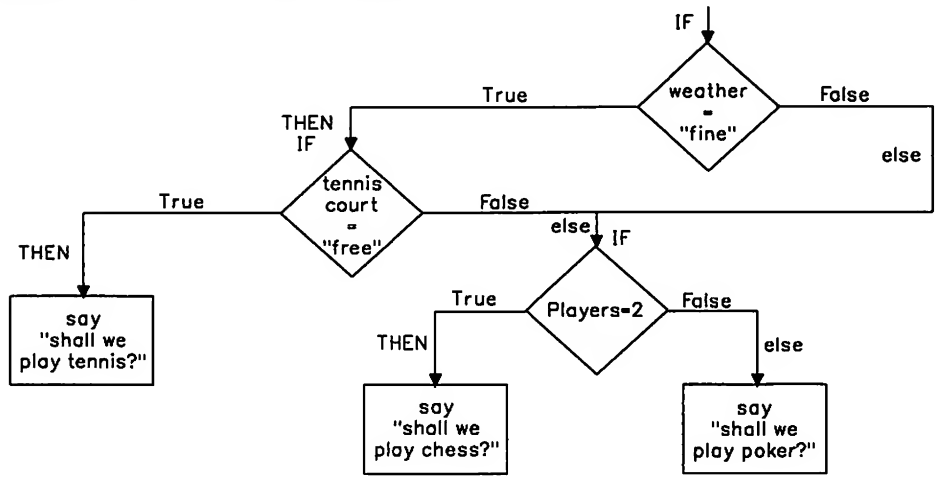
What will the program in Figure 6-19 do?

```
/* An example of a program that does not use "DO...END"  */
/* input data */
trace r
weather = "RAIN"
tenniscourt = "FREE"
players = 2

if weather = fine
then
   if tenniscourt = free
   then say "Shall we play tennis?"
   /* else say "Shall we take a stroll?" (DELETED)     */
else
   if players = 2
   then say "Shall we play chess?"
   else say "Shall we play poker?"
```

Figure 6-19. WHATTODO.CMD

Try it! The TRACE R (results) instruction at the beginning will help you see what is happening.

Answer: Remember the ELSE keyword is associated with the *nearest preceding* IF. The following flowchart shows what happens when these values are given for weather, tenniscourt, and players.



If the weather is anything but *fine*, the program displays nothing. It never considers how many players are available. The trace of WHATTODO.CMD program displays:

```
[C:\]whattodo
     3 *-*      weather = 'RAIN';
     3 >>>        "RAIN"
     4 *-*      tenniscourt = 'FREE';
     4 >>>        "FREE"
     5 *-*      players = 2;
     5 >>>        "2"
     7 *-*      If weather = 'FINE';
     7 +++        "0"

[C:\]
```

NOP Instruction

A THEN or ELSE keyword *must* be followed by an instruction. A semicolon is not sufficient. In cases where you intend that nothing should be done, use a NOP (no operation) instruction. You could use the NOP instruction to add an ELSE keyword in WHATTODO.CMD in Figure 6-19 on page 6-27. For example:

```
:
if weather = fine
then
   if tenniscourt = free
   then say "Shall we play tennis?"
   /* else say "Shall we take a stroll?" (DELETED)     */
   else NOP
else
   if players = 2
   then say "Shall we play chess?"
   else say "Shall we play poker?"
```

The following flowchart shows how the program flow is changed to.

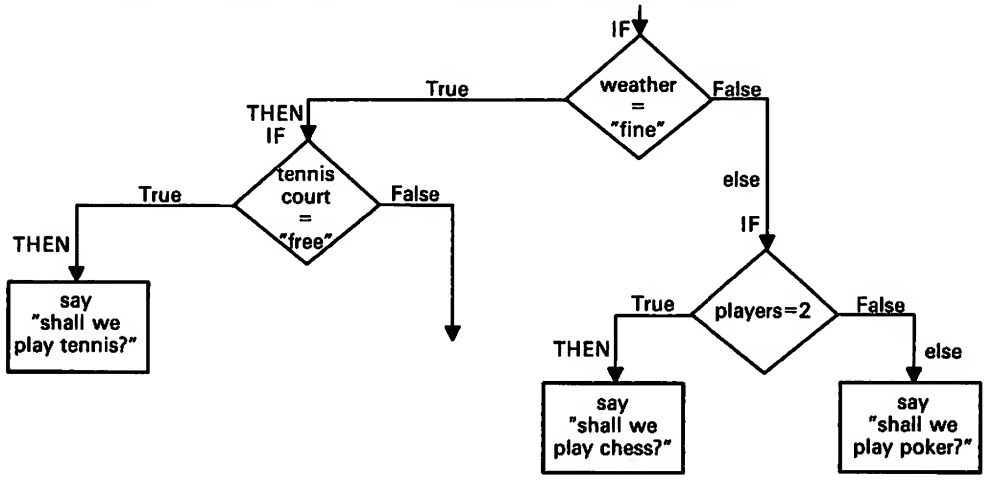


Figure 6-20 and Figure 6-21 on page 6-30 show examples using the NOP instruction.

```
/* Example: steering a course                              */

Say "Where is the harbor?"
pull where
select
   when where = "AHEAD" then nop
   when where = "PORT BOW" then say "Turn left"
   when where = "STARBOARD BOW" then say "Turn right"
   otherwise say "Not understood"
end
```

Figure 6-20. PILOT.CMD

```
/* Example: using NOP to simplify the presentation of   */
/* a set of conditions.                                 */

If gas = "FULL" & oil = "SAFE" & window = "CLEAN"
then nop
else say "Find a gas station!"
```

Figure 6-21. TRUCKER.CMD

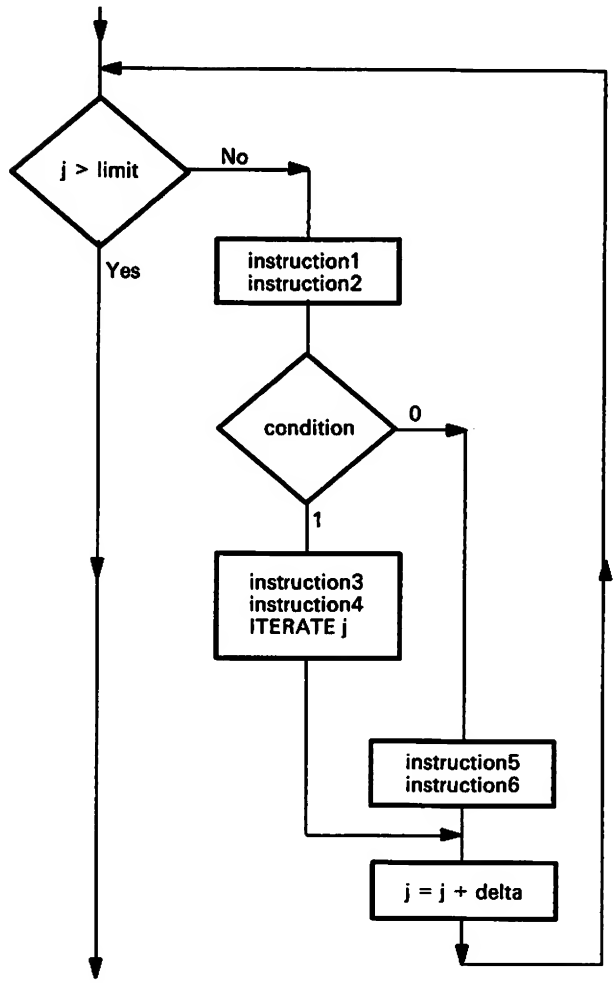
ITERATE Instruction

To bypass all remaining instructions in the loop and test the ending conditions, use the ITERATE instruction. Similar to LEAVE, ITERATE can be introduced by a THEN or ELSE keyword. Instead of leaving the loop altogether, REXX proceeds with the operations usually done at the bottom of the loop. If an UNTIL condition is specified, it is tested; if a counter is specified, it is incremented and tested; and if a WHILE condition is specified, it is tested.

If tests indicate that the loop is still active, then normal processing continues from the top of the loop. For example:

```
DO j = 1 to limit by delta
   instruction1
   instruction2
   if condition
   then do
      instruction3
      instruction4
      ITERATE j
   end
   instruction5
   instruction6
END
```

The following flowchart shows the program flow.



Compound DO Instructions

You can combine one repetitive phrase and one conditional phrase in a single DO instruction. You should know where in the loop the counters are updated and where the tests for leaving the loop are made (see "Conditional Loops" on page 6-12).

Compound DO instructions can do a lot of useful work. Figure 6-22 shows an example of how a simplified version of the POS() function may be implemented as a REXX function.

```
/* Example: the POSN() function is similar to the      */
/* POS(), except that the third argument ("start")     */
/* is not allowed                                      */

if arg() \= 2
then return              /* wrong number of arguments  */

if arg(1,omitted) | arg(2,omitted)
then return              /* argument was omitted       */

parse arg needle,haystack

last = length(haystack),  /* compute the rightmost     */
       -length(needle)+1  /* position that needle could */
                          /* be found in               */

do result = 1 to last,    /* Search for needle         */
   until substr(haystack,result,length(needle)) = needle
end
if result > last then result = 0
return result
```

Figure 6-22. POSN.CMD

Nested Loops

Sometimes a program is constructed of loops within loops. When you leave a loop, you may need to specify which loop you want to leave. To do this, give a DO loop a name (specify a counter in the DO instruction). If the loop does not contain a counter already, create one. For example:

```
DO outer = 1

⋮
END
```

This is the same, for all practical purposes, as DO FOREVER. In the previous example, outer is the counter for the loop. To leave a specific loop, put the name of its counter after the keyword LEAVE. For example:

```
DO outer = 1

⋮
   do until datatype(answer,WHOLE)
      say "Type a number. ",
          "When you have no more data, enter a blank line"
      pull answer
      if answer = "" then leave outer
   end

⋮
   /* process answer */
end
/* come here when there is no more data */
```

Chapter 7. Program Structure

This chapter discusses a different type of program-control—using instructions that run one program from within another. Using these *subsidiary* programs or *subroutines* can actually make programming easier.

Basics

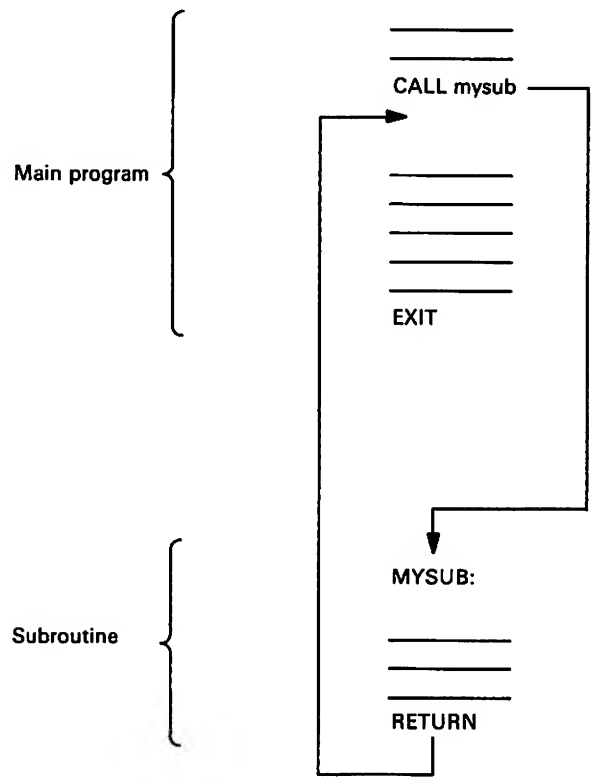
In this chapter:

Basics

- Subroutines
- External subroutines
- Using arguments
- Subroutines and data.

Subroutines

A *subroutine* is a segment of program code that can be called from more than one place in your main program. Subroutines can reside in the same file as the main program or they can reside in a separate REXX program file. The following diagram shows a subroutine that is in the same file as the main program.



A CALL instruction tells REXX to look through the program until it finds a corresponding *label*, a clause that marks the start of the subroutine.

REXX processes the instructions following the label until it encounters a RETURN instruction. RETURN tells REXX to resume processing in the main program, beginning with the instruction immediately after the CALL.

A subroutine can be called from more than one place in a program. That is, several CALL instructions can use the same subroutine. The subroutine always returns processing to the clause following the last CALL instruction.

Each CALL instruction can supply data, called *arguments*, which the subroutine can use when called. In the subroutine, you can determine the data supplied by using the ARG() function or the ARG instruction.

CALL Instruction

Figure 7-1 shows an example of how to have REXX run a subroutine at a particular point in a program:

```
CALL subname [argument1, argument2 ...]
```

where:

`subname` is the name of the subroutine. REXX searches first for the corresponding label in your program. A label consists of a symbol followed by a colon (:). For example:

```
subname:
```

If no such label is found, REXX looks for a built-in function or program file named subname. (See the search order in "Comparing Subroutines and Functions" on page 7-13.)

`argument` is any data that you want passed to the subroutine. The subroutine can collect the arguments with the ARG instruction or the ARG() function.

Figure 7-1 shows a program that displays the squares of numbers from 1 to 5. The calculation is performed in a subroutine.

```
/* Simple example of using CALL instruction */
trace r                              /* we turn on tracing    */
                                     /* so you can see the    */
                                     /* subroutine in action  */

say "This is the main program"         /* remark 1            */
do num = 1 to 5
   call square                         /* calls subroutine    */
   say "Back in the main program."     /* remark 3            */
   say num "squared is" num2           /* display result      */
end
exit                                   /* end the program     */

square:                          /* subroutine begins   */
say "This is the subroutine."          /* remark 2            */
num2 = num * num                       /* calculate square    */
return                                 /* resume main program */
```

Figure 7-1. SQUARIT.CMD

Try this program without the TRACE instruction and extra remarks.

RETURN Instruction

The RETURN instruction takes processing back to the main routine. Processing continues with the instruction following the last CALL. The full form of the instruction is:

```
RETURN [expression]
```

where, if expression is specified, it is assigned to the REXX special variable, RESULT, which can then be used by the main program. But, if expression is omitted, RESULT is *dropped* and not assigned a value. Its value is its own name—RESULT.

Test Yourself

Figure 7-2 shows a program that simulates a children's race game that used to be played with dice. Write the subroutine TELL to tell who is winning.

```
/* Example of a subroutine: a child's race game     */
a = 0                     /* Arthur starts from zero */
b = 3                 /* Barry gets a headstart of 3 */
do 15
   a = a + random(1,6)   /* Arthur gets first turn  */
   call tell             /* Who's ahead now         */
   b = b + random(1,6)  /* Now it is Barry's turn   */
   call tell             /* Who's ahead now         */
end
exit                     /* End of main program     */
```

Figure 7-2. RACEGAME.CMD

Copy the main program and your subroutine into another file and test your program.

Answer: Figure 7-3 is an example of a possible solution.

```
/*-------------------------------------------------*/
/* Subroutine to display the position              */
/* ================================              */
/* INPUT: a (Arthur's score)                       */
/*        b (Barry's score)                        */
/* RESULT: displayed on user's screen              */
/*-------------------------------------------------*/
TELL:
values = "Arthur =" a ";  Barry =" b "; "
select
   when a > b then say values "Arthur is ahead"
   when b > a then say values "Barry is ahead"
   otherwise say values "Neck and neck!"
end
return
```

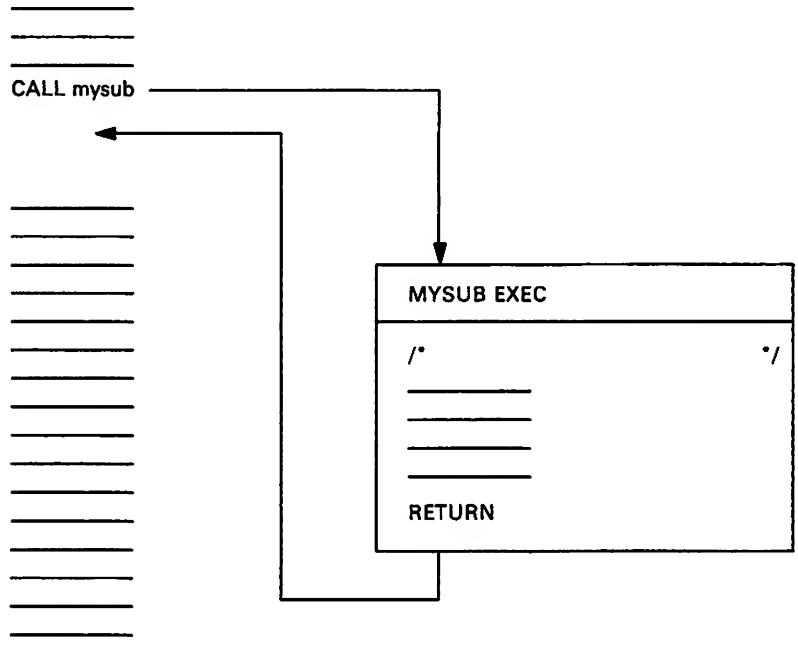
Figure 7-3. RACEGAME.CMD Subroutine

In this sample solution, there are no arguments on the CALL instruction. Nevertheless, a person reading the program needs to know what data the subroutine is using.

A well-designed subroutine operates on a clearly defined set of data. To make your program more readable, you should define this data in comments at the beginning of the subroutine.

External Subroutines

The subroutines that have been discussed are internal routines. Subroutines can also exist as a separate REXX program file. The following diagram shows a subroutine that is a separate REXX program file.



In an *external* routine, the variables belonging to the calling routine are *not* available to the subroutine. Therefore, data:

- Must be formally passed to the subroutine as arguments to the CALL instruction.
- Can only be returned to the caller by assigning a value to the variable RESULT, using the RETURN instruction. If necessary, the calling routine can then parse the RESULT value into a number of variables.

Note: The variables of the CALL are available to an *internal* subroutine unless you use the PROCEDURE instruction (see "PROCEDURE Instruction" on page 3-16).

Using Arguments

In the example shown in Figure 7-4, type the program name, add, followed by the two numbers to be added. The numbers are assigned to variables first and second by the ARG instruction. Information given to a program in this manner is called an *argument*. That is, the numbers given ADD.CMD to add together are arguments to the startup command, ADD.

```
/* the sum of two numbers, this time    */
/* typed at the command prompt          */
arg first second     /*collects entries */
say "The sum is" first + second
```

Figure 7-4. ADD.CMD

In much the same way, you can provide a subroutine with the needed information by arguments passed by the CALL instruction that starts the subroutine. To assign the arguments to variables, you can use the ARG instruction or the PARSE ARG instruction.

The difference is:

ARG assigns argument data to variables, translating lowercase letters into uppercase. In this way, ARG is similar to PULL. ARG is the short form of the instruction PARSE UPPER ARG.

PARSE ARG assigns the information to variables exactly as it is entered, with no translation to uppercase.

For example, here is a CALL instruction:

```
CALL BAKE "white", "fresh", "sweet", "dessert"
```

If you want the results of these four expressions assigned to flour, butter, sugar, and cookies, you would write:

```
PARSE ARG flour, butter, sugar, cookies
```

But, if you wanted the four arguments to be translated to uppercase, you would write:

```
ARG flour, butter, sugar, cookies
```

As there are commas between the expressions in the CALL instruction, there are likewise commas between the symbols in the PARSE ARG or ARG instruction when it is used in this way. For example, the instruction:

```
CALL words "a string of words",5
```

might be parsed using:

```
WORDS:
PARSE ARG first second third fourth rest, number
```

The result would be that:

```
first  gets a
second gets string
third  gets of
fourth gets words
rest   gets  (blank)
number gets 5
```

Figure 7-5 is an example of the main program, that shows how:

- CALL passes arguments to a subroutine.
- ARG assigns the argument values to variables.
- RETURN assigns a value to the RESULT variable.
- RESULT is used by the main program.

```
/* Main program to gather input and display result     */
Say "To calculate the material you need to make a box,"

/* Input the dimensions of the desired box (meters)    */

say "type the desired length of the box:"
pull length

say "type the desired width:"
pull width

say "type the height:"
pull height

/* call the subroutine program BOX.CMD  with arguments */

CALL box length, width, height

/* report the returned value from the RESULT variable   */
SAY 'Material required =' RESULT 'square meters'
exit
```

Figure 7-5. MAKEBOX.CMD

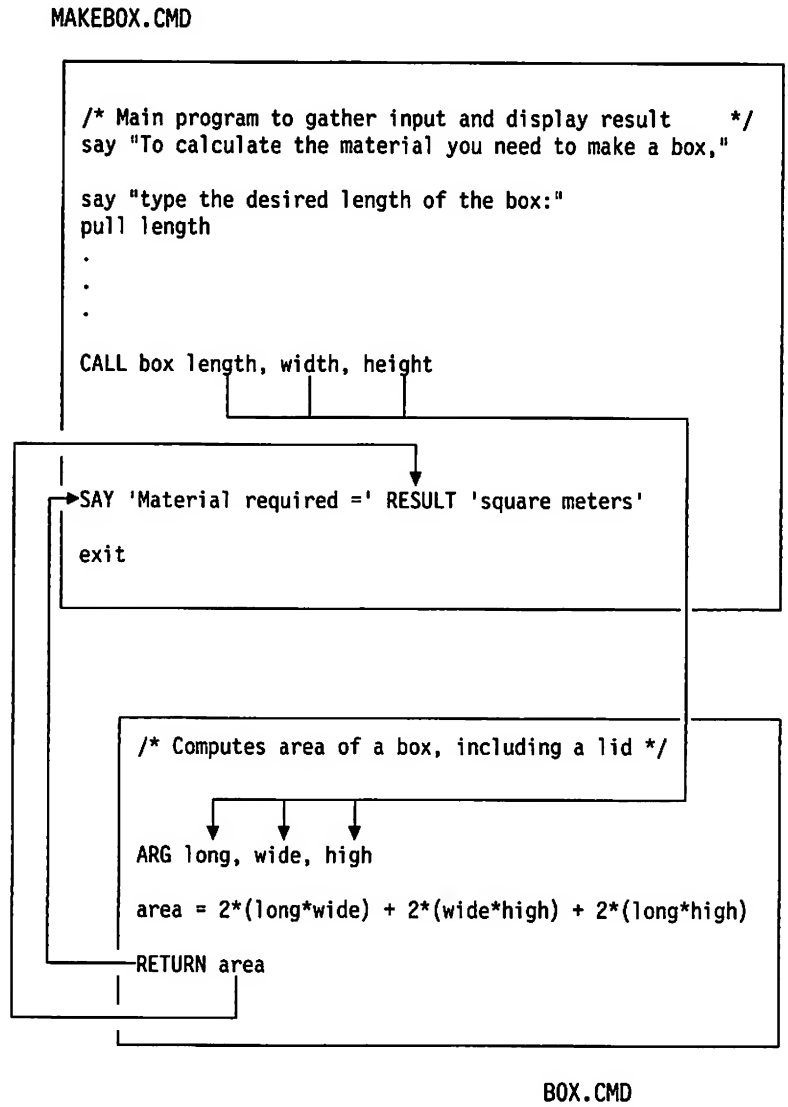
Figure 7-6 is the subroutine program called by MAKEBOX.CMD.

```
/* computes area of a box */
/* including a lid        */

ARG long, wide, high

/* total area is base and top plus  */
/* short sides plus long sides      */

area = 2*(long*wide) + 2*(wide*high) + 2*(long*high)

RETURN area
```

Figure 7-6. BOX.CMD

When you run MAKEBOX, the data you enter is gathered by the PULL instructions and then passed, as arguments of CALL, to BOX.CMD. The calculation of the area of the box is then passed back to MAKEBOX.CMD by the RETURN instruction to the variable RESULT. Processing of MAKEBOX then resumes with the next clause following the CALL.

The following diagram shows the flow of data between the two programs.



If program variables are referred to by the same names both outside and inside an *internal* routine (a routine that exists in the same file as the CALL instruction), then it is not necessary to include them as arguments on the CALL or ARG instructions. However, not including them could make it more difficult for a user reading your program to understand what your subroutine does. So it is a good idea to give a list of the arguments in the comments that introduce the subroutine.

ARG() Function

Another way to pass arguments to a subroutine is to use the ARG function. For example:

```
CALL subname [argument1, argument2 ...]
```

where:

`subname`	is the name of the subroutine.
`argument1, argument2,...`	are expressions. The value of each is computed and can be obtained in the subroutine by using the ARG() function. ARG(1) returns the first argument ARG(2) returns the second argument and so on... You can have up to 20 arguments on a CALL instruction.

Figure 7-7 shows an example that calls a subroutine using arguments.

```
/* Example: calling a subroutine with arguments */
do 3
  call triple "R"
  call triple "E"
  call triple "X"
  call triple "X"
  say
end
say "R...!"
say "E...!"
say "X...!"
say "X...!"
say
say "REXX!"
exit                                /* end of main program */
/*-------------------------------------------------------*/
/* Subroutine to repeat a shout three times              */
/* ======================================              */
/* The first argument is displayed on the screen, three */
/* times on one line, with suitable punctuation.         */
/*-------------------------------------------------------*/
TRIPLE:
say arg(1)", "arg(1)", "arg(1)"!"
return
```

Figure 7-7. CHEER.CMD

The following is displayed on the screen when you run the program.

```
cheer
R,  R,  R!
E,  E,  E!
X,  X,  X!
X,  X,  X!

R,  R,  R!
E,  E,  E!
X,  X,  X!
X,  X,  X!

R,  R,  R!
E,  E,  E!
X,  X,  X!
X,  X,  X!

R...!
E...!
X...!
X...!

REXX!
```

The EXIT instruction in Figure 7-7 on page 7-8 stops the main program from running on into the subroutine.

Subroutines and Data

Passing arguments to a subroutine is one form of parsing input information. This is a particularly important concept in REXX.

The best starting point for a large program is to study the information that you want REXX to work with. The next few chapters concentrate on how to analyze and calculate data and how REXX shares information with other programs and devices.

Summary

This completes "Basics" in this chapter. You have learned how to:

- Use subroutines with the CALL and RETURN instructions
- Use subroutines in the same program file as the main program or in a separate file
- Pass data to a subroutine, using the ARG instruction and the ARG function.

"Advanced Topics" in this chapter discusses advanced elements of program structure, including:

- Modular programming
- Creating your own functions
- Jumps and condition traps.

To continue with "Basics," go to page 8-1.

Advanced Topics

In this chapter:

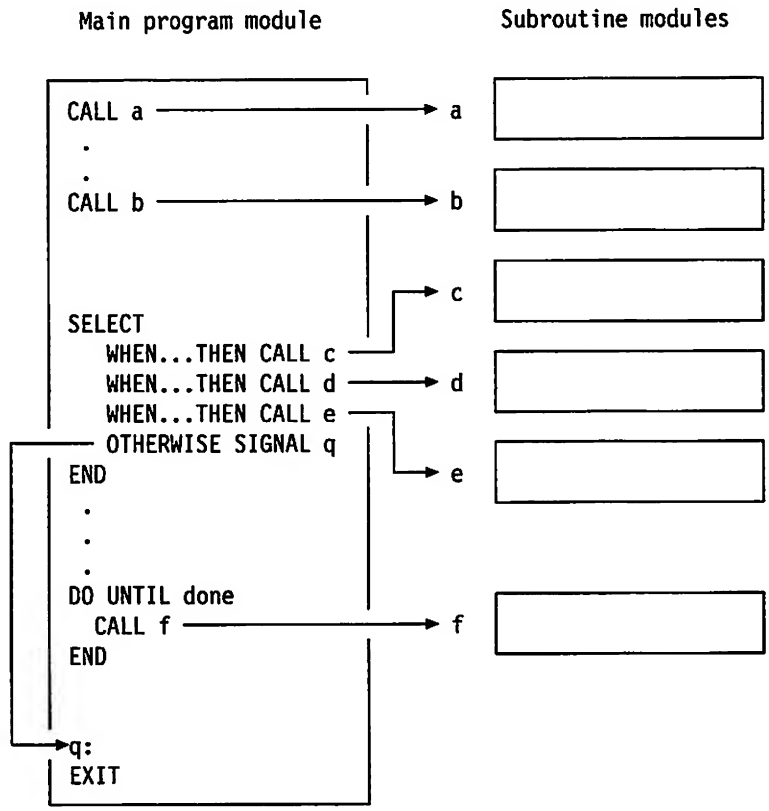
Advanced Topics

- Structured programming
- Function calls
- Comparing subroutines and functions
- Jumps
- Condition traps.

Structured Programming

Using the CALL instruction to call a subroutine is part of an approach to programming called *structured programming*.

Experienced programmers rarely write complex programs as a single list of instructions. Rather, they separate a large job into smaller units called *modules*. Then, they create a single *main module* that calls all the others, either in a specific listed order or by means of control instructions such as IF, SELECT, and DO. For example:



Structured programming helps programmers in the following ways:

- It makes planning a program easier because large projects can be separated into smaller tasks that are easier to understand.

- It makes the finished program more readable by users. This is a definite help when correcting and improving the program. You can fix and enhance one module at a time. Errors are easier to trace, and the overall structure is clearer.
- As you become more fluent in the REXX language, you become more adept at defining the work a program can do. You can define clearly what a module should do before you write a line of program code. For example, the program code you write will have applications beyond the task it was written for. A clever routine for sorting a directory might prove useful in another file-management program.

For more information about structured programming, refer to Chapter 11, "Program Style."

Function Calls

A subroutine can produce a computed result, or *return value*, that the calling procedure can use. The RETURN instruction does this (see "RETURN Instruction" on page 7-3). You have the following options:

- Use CALL to call the subroutine and then get the return value from the built-in variable RESULT.
- Call the subroutine as a function call, the same way that you call the REXX built-in functions.

Creating a Function

A subroutine called by a function call is essentially no different than any other subroutine that returns a value. That is an important distinction, because the very definition of a function is that it *always returns a value*, even if that value is a null string.

Arguments of the function (the input that goes between the parentheses in the function call), can be read with the ARG instruction or with the ARG function.

Figure 7-8 shows a subroutine that calculates and returns the average number of letters per word.

```
/* Returns the average letters per word in an input line */
arg line                          /* get the line; set TOTAL to 0 */
total = 0

do num = 1 until line = ""        /* repeat until LINE is empty,  */
                                  /* adding 1 to NUM with each    */
                                  /* iteration                    */

   parse var line word line       /* take out just the first word */
                                  /* remaining in LINE            */

   total = total + length(word) /* add its length to TOTAL       */

   end                            /* see if LINE is empty yet...  */
                                  /* if so, the loop ends.

avg = format(total/num,,0)        /* now, divide TOTAL (letters)  */
                                  /* by NUM(ber of words) to get  */
                                  /* the average per word         */

return avg                        /* return the number AVG        */
```

Figure 7-8. WORDAVG.CMD

You can also create functions that return non-numeric values, as shown in Figure 7-9.

```
/* Returns today's date in 'natural language' format */

/* First, use DATE() function and its options to...        */
month = date(m)                       /* ...get the month   */
day = date(w)                         /* ...get the weekday */
parse value date() with cdate . .     /* ...get the date    */

/* The RIGHT() function returns the last digit of the date */
/* so we can add on the proper suffix (similar to the way  */
/* we used for Figure 3-9 on page 3-11.*/
select
   when right(cdate,1) = 1 then th = "st"
   when right(cdate,1) = 2 then th = "nd"
   when right(cdate,1) = 3 then th = "rd"
   otherwise th = "th"
end

return day"," month cdate||th          /* return the date */
```

Figure 7-9. DATESTMP.CMD

Comparing Subroutines and Functions

There are differences and similarities between subroutines and functions.

The differences are:

- To call a subroutine, you use a CALL instruction:

  ```
  CALL routine [argument1, ... ]
  ```

  To call a function, you use a function call:

  ```
  routine([argument1, ... ])
  ```

- A subroutine need not return a result, but a function *must* return a result.

  In a subroutine, you can write:

  ```
  RETURN
  ```

  In a function you must at least write:

  ```
  RETURN ""   /* This returns a null string */
  ```

- A subroutine sets the value of the special variable RESULT. The result returned by a function is used in the expression where the function call appeared.

The similarities are that both:

- Use the ARG and PARSE ARG instructions and the ARG() function for obtaining the values of their arguments.
- Can be *internal* (starting with a label in the same file as the CALL instruction or the function call) or *external* (located in a different file).
- Have the same *search order*. When a call to `routine` is recognized:
  1. REXX first looks for the label `routine:` in the same file.
  2. If no label by that name is found, REXX looks for a built-in function called `routine()`.
  3. If none of its own functions have that name, REXX looks for an external routine; that is, a program in a file named `routine`.

There are many kinds of external routines that REXX can use, including those written in other languages. These are made available to REXX through *function packages*, which are described in the *REXX Reference*.

Using a Call of the Other Kind

There is another similarity between functions and subroutines. Where convenient, programs designed as functions can be called as subroutines. If they always return a result, programs designed as subroutines can be called as functions. Both, when they are *internal*, can use the PROCEDURE instruction.

You could eliminate the CALL instruction in MAKEBOX.CMD (see Figure 7-5 on page 7-6) and simply call the subroutine BOX.CMD (see Figure 7-6 on page 7-6) as a function in the final SAY instruction. For example:

```
say "Material required =" box(length,width,height) "sq. meters"
```

In the following example, the POS() function returns the character position of a substring (`needle`) within a larger string (`haystack`).

```
POS(needle,haystack)
```

Figure 7-10 shows an example that is called as a subroutine.

```
/* to remove NEEDLEs from haystack */
do forever
   call pos needle,haystack
   if result = 0
   then leave
   else haystack = delstr(haystack,result,length(needle))
end
```

Figure 7-10. NEEDLE.CMD

This routine removes every instance of one string within another. It uses the DELSTR() and LENGTH() functions, which are also built into REXX, and are called here as functions. In this example:

1. POS(), called as a subroutine, searches in the string stored in `haystack` for the first occurrence of the string in `needle`.
2. That character position (a number) is then assigned to the RESULT variable. If no occurrence if found, the loop ends.
3. DELSTR() then removes the `needle` substring from `haystack`, deleting from the RESULT starting position, for the length of `needle` (determined by the LENGTH() function).

Jumps

A *jump* shifts REXX processing to a different point in the program. After the jump, the program does not automatically return to the calling instruction.

SIGNAL Instruction

The SIGNAL instruction can jump (transfer control) to another part of your program. SIGNAL is a one-way path. When REXX encounters a SIGNAL instruction in the middle of a program, any SELECT constructs or DO loops it has been processing are abandoned. Unlike CALL, SIGNAL cannot be used to jump back into or jump around within a DO loop. This means that you should use SIGNAL only to bring your program to an exit. For other purposes, it is better to manipulate program control using IF, SELECT, or DO.

To tell REXX to go to another part of the same file, use the instruction:

```
SIGNAL symbol
```

This causes a jump to the specified label, the `symbol` followed by a colon (:). REXX searches from the top of the file for the clause `symbol:` and processing continues from there.

The SIGNAL instruction always stores its line number in the REXX special variable SIGL.

This is an example of an *abnormal end* to a program using SIGNAL.

```
SIGNAL abend
:
EXIT                              /* end of ordinary code */
/*-------------------------------------------------------*/
/* This code handles abnormal ends                       */
/*-------------------------------------------------------*/
ABEND:
say "Abnormal end signalled at line" sigl,
 ||". Cannot continue."
exit
```

The first EXIT instruction is there to stop the normal program from running on into the abend: (abnormal end) routine.

SIGNAL is not a *go to* statement similar to the BASIC instruction GOTO, because SIGNAL causes the logic of other control structures to be abandoned. The concept of *go to* does not really apply to a structured language such as REXX. Most tasks or routines that are to be performed according to some condition are better handled as subroutines. They are easier to read that way, and they are easier to correct.

The exception is when you use SIGNAL to call an exit routine. This is a specific application of SIGNAL, to act as a *detector* to bring a program to an end whenever a specific condition occurs. In other words, you can set a *trap*.

Condition Traps

The SIGNAL instruction by itself initiates a jump from one particular point in a program (the SIGNAL clause) to another, indicated by a label.

Another way to use SIGNAL is with the keyword ON and the name of a specific *condition* that you want REXX to test for. Whenever the specified condition is detected, processing immediately jumps to the corresponding label. For example:

```
SIGNAL ON condition [NAME trapname]
```

When a SIGNAL ON instruction is in effect (*enabled*) and the given `condition` occurs, REXX jumps to a label that can be either:

- The name of the `condition` itself (the default)
- An optional `trapname`, specified by the `NAME` keyword.

This trap remains enabled for the rest of the program or until issued the instruction:

```
SIGNAL OFF condition
```

These are the conditions that can be trapped by SIGNAL ON instruction:

`ERROR` — Sets a trap to a subroutine with the label `ERROR:`. An ERROR condition occurs whenever a REXX program issues a command (a string expression passed to the environment) that results in an error in the default environment. This includes commands that produce nonzero return codes and commands that are unknown to the system. (This includes failures only when a FAILURE condition trap is not set.)

FAILURE	Sets a trap to a subroutine with the label FAILURE:. A FAILURE condition occurs whenever a REXX program issues a command to the default environment and the system encounters a severe error preventing it from processing the command. Refer to "Trapping Command Errors" on page 5-12 for more information.
NOTREADY	Sets a trap to a subroutine with the label NOTREADY:. A NOTREADY condition occurs whenever an error occurs during an input or output operation. (Refer to Chapter 10, "Input and Output," for more information.)
NOVALUE	Sets a trap to a subroutine with the label NOVALUE:. A NOVALUE condition occurs whenever a symbol that could be the name of a variable is encountered and the variable does not exist. This can be especially useful for finding misspelled variable names.
SYNTAX	Sets a trap to a subroutine with the label SYNTAX:. A SYNTAX condition occurs whenever a REXX syntax error is detected. This is especially useful for debugging programs in which syntax errors occur, but the user is unable to provide an accurate description of the problem.

If the trap you have set with SIGNAL ON detects one of the previous conditions:

1. REXX stops whatever instruction it is processing.
2. The line number of that instruction is assigned to the special variable SIGL.
3. The condition is *disabled*; it is set to SIGNAL OFF.
4. Processing then jumps (as in a normal SIGNAL) to the appropriate label (to `trapname:`, if given; otherwise, to `condition:`).

Using CALL ON

In certain instances, it is possible to resume program processing after a condition is trapped. In such an instance, use the instruction:

```
CALL ON condition [NAME trapname]
```

See "Trapping Command Errors" on page 5-12 for an example comparing the CALL ON and SIGNAL ON instructions.

Useful Functions

The following two functions you may find useful:

CONDITION()	Returns the keyword (CALL or SIGNAL) used to trap the condition. Used with its options, CONDITION() can also supply the name of the trapped condition (for example, ERROR), its status (ON, OFF, or DELAY), or an associated description.
VALUE()	Returns or sets the value of a variable whose name may be a string expression. VALUE() does *not* trigger a NOVALUE condition.

For more information about these instructions and functions, refer to "Conditions and Condition Traps" in the *REXX Reference*.

Chapter 8. Parsing

Parsing is a way of analyzing information for your program to use. It is one of the most powerful and practical features of the REXX language. You may have noticed that it has already been used in many examples. This chapter discusses these ideas and examines the parsing options one by one.

Parsing is separating input data and assigning it to one or more variables. Some examples of sources of the input data are:

- The keyboard—more specifically, any data a user types in while a program is running
- An *input parameter*—an option typed after the command that starts a program
- An external data source—such as a *queue* or a *disk file*.

Basics

In this chapter:

Basics

- Conversations
- Parsing variables and expressions
- Specialties.

Conversations

The most common source of input data is the person using the program. The following is a review of the instructions you use to *converse* with the user of your program—that is, to allow that person to direct the program's processing.

Prompting the User for Input

To display information on the screen, use `SAY expression`. The `expression` is evaluated and the result is displayed as a new line on the screen. For example, `say 3 * 4 "= twelve"` causes the following to be displayed on the screen.

```
12 = twelve
```

If you want to display a clause that occupies more than one line in your program, use a comma at the end of a line to indicate that the expression continues on the next line. For example, the instruction:

```
say "What can not be done today, will have to be put off",
    "until tomorrow."
```

causes the following to be displayed on the screen:

```
What can not be done today, will have to be put off until tomorrow.
```

The continuation comma is replaced by a blank when the expression is displayed. Remember that the continuation comma cannot be enclosed in quotes or REXX will consider it part of the string.

Having asked the user a question using SAY, you can collect the answer using PULL. When `pull symbol` is processed, the program pauses for the user to type data on the command line and press the Enter key. Whatever the user types is translated to uppercase and then assigned to the variable `symbol`.

The PULL instruction is a short version of the instruction PARSE UPPER PULL, which converts lowercase letters in the user input to uppercase. The program recognizes a user's response whether it is in uppercase, lowercase, or mixed case. To get the data as it is, without this conversion, use the form:

```
PARSE PULL symbol
```

Figure 8-1 shows an example that uses both PULL and PARSE PULL.

```
/* Another conversation */
say "Hello!  What is your name?"
parse pull name
say "Say," name", are you going to the party?"
pull answer
if answer = "YES"
then say "Good.  See you there!"
```

Figure 8-1. CHITCHAT.CMD

The user's name is displayed exactly as it was typed, but answer is translated to uppercase. This simplifies the program by ensuring that the same action is taken regardless of the way the user types the word *yes*.

Test Yourself

1. Figure 8-2 shows a program that asks a question.

```
/* Simple question (?) */
say "Mary, Mary, quite contrary"
say "How many letters in that?"
pull ans
if ans = length(that)
then say "Quite right!"
else say "Oh!"
```

Figure 8-2. RIDDLE.CMD

What is displayed on the screen, if the user responds:

- 21
- 4
- Four.

2. What is displayed on the screen when you run the program shown in Figure 8-3?

```
/* Example: expressions that continue for more      */
/* than one line.                                   */
x = 3
say "x =" x
say
say "Ham,",
    "Shem",
    "and Japheth"
say "Silly"
    "Billy"
```

Figure 8-3. NOAH.CMD

3. Create a file called PULLIN.CMD, type the program shown in Figure 8-4, and try to run the program.

```
/* Example: appending input, using PULL,            */
/* to a REXX variable                               */
text = ""
do until input = "QUIT"
   say "Text so far is:"
   say text
   say "Would you like to add to that?",
       " If so, type your message.",
       " If not, type QUIT."
   pull input
   text = text||input
end
```

Figure 8-4. PULLIN.CMD

Did the program run correctly? If not, study the error messages and make sure you copied everything correctly. Notice that:

- When you run the program, everything you type is changed to uppercase letters.
- There are not any blanks between the *old* text and the *new* input.

4. Change `pull input` to `parse pull input`. Change the concatenate operator "`||`" to a single blank, then try the program again. Notice that:
   - Your input does not get changed to uppercase.
   - There is always one blank between the *old* text and the *new* input.
   - You cannot exit the program by typing `quit`, but you *can* exit by typing `QUIT`.

Answers:

1. This is displayed on the screen:
   - `Oh!`
   - `Quite right!`
   - `Oh!`
2. This is displayed on the screen:

```
[C:\]noah
x = 3

Ham, Shem and Japheth
Silly

[C:\]BILLY
SYS1041: The name specified is not recognized as an
internal or external command, operable program or batch file.
[C:\]
```

Because there is no comma after `Silly`, `Billy` is treated as a command. If no such command exists, then the OS/2 program issues an error message.

Getting Data When the Program Starts

When you want to run your program, type its file name at the command prompt. This can be followed by information for the program to use in the form of arguments. To use these arguments in the program, use the ARG instruction. ARG parses command arguments in the same way that PULL parses data from the keyboard, except that the first word typed (the name of the program) is ignored.

Figure 8-5 shows an example of how a parsed argument can be used in a SELECT instruction.

```
/* telephone tickler: displays the phone number */
/* of a person whose NAME is given as the       */
/* argument; e.g., the command "phone anne"     */
/* displays "ANNE'S NUMBER IS 555-3434"         */

arg name                          /* get the name */

select
   when name = "WILLIAM"  then
      number = "555-1212"
   when name = "ANNE"     then
      number = "555-3434"
   when name = "LOUISE"   then
      number = "555-5656"
   when name = "STEVE"    then
      number = "555-7878"
   when name = "HELEN"    then
      number = "555-9090"

   otherwise
      say "I do not have that number."
      exit
end

say name||"'S NUMBER IS" number||"."
exit
```

Figure 8-5. PHONE.CMD

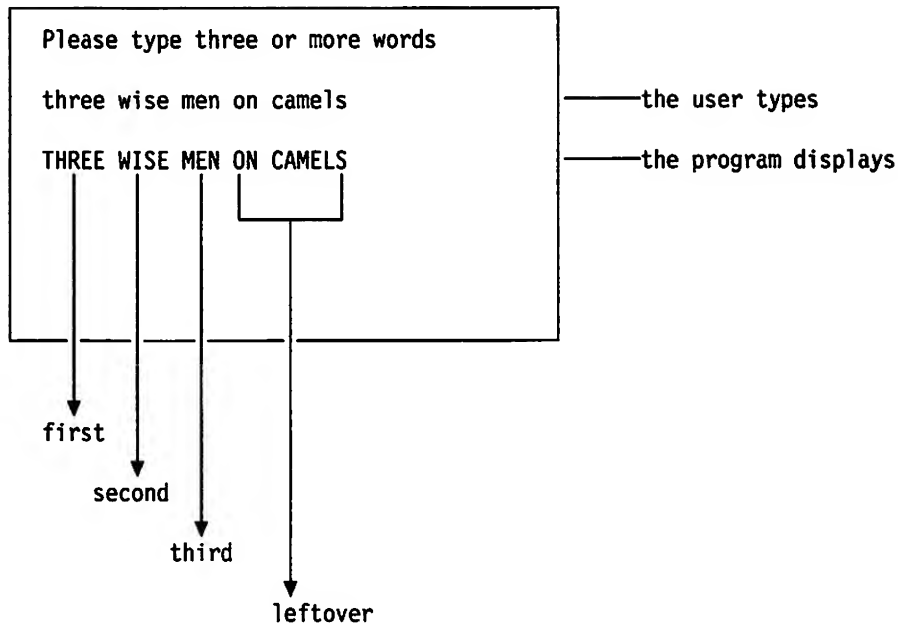
The ARG instruction is the short form of the PARSE UPPER ARG instruction. To obtain arguments without the uppercase translation of letters, use the PARSE ARG instruction instead.

Multiple-Variable Assignment

PULL and ARG can also *fetch* each word into a different variable. In the following example, `first`, `second`, `third`, and `leftover` have been chosen as the names of variables.

```
say "Please enter three or more words:"
pull first second third leftover
⋮
```

The following is displayed on the screen.



The program pauses and the user can type something on the command line. When the user presses the Enter key, the program continues and the variable:

first is given the value THREE.
second is given the value WISE.
third is given the value MEN.
leftover is given the value ON CAMELS.

In general, each variable gets a word (without blanks) and the last variable gets the rest of the input, if any (with blanks). If there are more variables than words, the extra variables are assigned the null value.

The same thing can be done with ARG. Figure 8-6 shows a program that accepts user input and then displays it in a different order.

```
/* Example: this program starts by assigning the words     */
/* from the command line to REXX variables and then        */
/* displays them, swapping the first and third arguments. */

arg first second third rest
say third second first rest
```

Figure 8-6. MIX.CMD

The following is displayed on the screen.

```
[C:\] mix james joe jack and jeff
JACK JOE JAMES AND JEFF
```

When the ARG instruction is processed, the variable:

first is given the value JAMES.
second is given the value JOE.
third is given the value JACK.
rest is given the value AND JEFF.

You can then use the SAY instruction to display the variables in any order you choose.

Checking for Input Errors

To ensure that the user types in the right number of words, provide one extra variable and test that it is empty. Also, test the variable that holds the last word that the user is expected to type. By testing both variables for a null value, you can be sure that each of your variables contains exactly one word.

Figure 8-7 shows an example that ensures that the user has typed the correct number of words.

```
/* Example: getting the number of words that you want   */

good = 0
do until good
   say "Please type exactly three words"
   pull first second third rest
   select
      when third = "" then say "Not enough words"
      when rest \= "" then say "Too many words"
      otherwise good = 1
   end
end
```

Figure 8-7. FUSSY.CMD

Using a Placeholder

The period symbol (.) by itself may not be used as a name, but it may be used as a place-holder with the PULL instruction. For example, `pull . . lastname .`, would discard the first two words, assign the third word into `lastname`, and discard the remainder of the input.

Test Yourself

1. What is displayed on the screen when you run the program shown in Figure 8-8?

```
/* Example: the PULL instruction */
Say "Where did Jack and Jill go?"
parse pull one two three four five six .

       /* User replies 'To fetch a pail of water' */

say one two six
say
Say "Will you buy me a diamond ring?"
pull reply .

        /* User replies 'Yes, if I can afford it' */

say reply
```

Figure 8-8. PULLING.CMD

2. Write a program that asks users for their name and then greets them by first name only. Your program should ignore any other names.

Answers:

1. The following is displayed on the screen.

```
Where did Jack and Jill go?
To fetch a pail of water
To fetch water

Will you buy me a diamond ring?
Yes, if I can afford it
YES,
```

2. Figure 8-9 shows an example of a possible answer.

```
/* Example: selecting a single word */
say "Howdy!  Say, what is your name?"

pull reply .              /* The period causes second */
                          /* and subsequent words to  */
                          /* be ignored               */

say "Pleased to meet you," reply
```

Figure 8-9. HOWDY.CMD

Parsing Variables and Expressions

PULL and ARG are the short versions of options of the PARSE instruction. As well as parsing replies from the user and the data from the command line, you can also parse:

- The values of variables by using `PARSE VAR symbol v1 v2 v3 ...`
- The results of expressions by using `PARSE VALUE expression WITH v1 v2 v3 ...`

Figure 8-10 shows an example of parsing variables and expressions.

```
/* Examples of parsing variables and expressions   */
phrase = "Three blind mice "
PARSE VAR phrase number adjective noun
say number                    /* says 'Three'       */
say adjective                 /* says 'blind'       */
say noun                      /* says 'mice'        */
PARSE VALUE copies(phrase,2) WITH . a . b . c
say b a c               /* says 'Three blind mice' */

/* Here is a very useful trick for taking */
/* the first word away from a sentence    */
PARSE VAR phrase first phrase
say first                     /* says 'Three'       */
say phrase                    /* says 'blind mice'  */
```

Figure 8-10. PARSING.CMD

You can use parsing to analyze data from sources other than user input, such as from files, queues, or hardware devices. For more information, see Chapter 10, "Input and Output."

Parsing Numeric Data

Only parsing text data has been discussed, but all of the parsing options that have been described work equally well with numeric data.

The way REXX handles numbers is another measure of its flexibility. Most computer languages have many rules about the differences between text data and number data. In REXX, a number is simply a string that can be calculated.

Summary

This completes "Basics" in this chapter. You have learned how to:

- Prompt, receive, and check user input
- Manipulate variable templates
- Use the period as a placeholder
- Parse variables and expressions.

"Advanced Topics" in this chapter discusses parsing using:

- Literal-string patterns
- Character positions
- Variables in patterns
- String functions.

To continue with "Basics," go to page 9-1.

Advanced Topics

In this chapter:

Advanced Topics

- Parsing with patterns
- Literal string patterns
- Character position
- Variables in patterns
- String functions.

Parsing with Patterns

By using *patterns* in a parsing template, you can have your programs analyze all types of information. In this chapter, several kinds of parsing patterns are discussed.

For more information about the use of patterns refer to "Parsing" in the *REXX Reference*.

Literal String Patterns

One way to parse data is by *literal pattern*. If your PARSE instruction template specifies a literal string (one or more characters enclosed in quotes), the data being parsed is split at the point where the string is found.

Figure 8-11 shows an example where the ARG instruction parses the data given with the command TAKE. The first literal pattern is the first / and the second literal pattern is the second /.

```
/* Example: recognizing options */

arg drink form shelf  "/"  typ1 typ2 typ3 "/" rest
say drink form shelf":" typ1 typ2 typ3 "("rest")"
:
```

Figure 8-11. TAKE.CMD

The following is displayed on the screen.

```
[C:\] take coffee beans /kenya decaf/in bags
COFFEE BEANS : KENYA DECAF  (IN BAGS)
```

When the ARG instruction is processed:

- The words in front of the first pattern are parsed in the usual way into `drink`, `form`, and `shelf`. In this example, `shelf` is set to null; that is, an empty string or `""`. The SAY instruction keeps the space following the variable name. That is why there is a space before the colon.
- The words between the first pattern and the second pattern (if there is one) are parsed into `typ1`, `typ2`, and `typ3`. Here, `typ3` is set to null, but SAY still displays the extra space before the parenthesis.

- If there is a second pattern, the words that follow it are assigned to the variable rest. In this example, rest is assigned the two-word string IN BAGS.

This technique of parsing using literal patterns can be used with any of the parsing instructions.

Character Position

Another way to parse a string is by the *position* of the individual characters. There are two types of positions, absolute and relative.

Absolute Position

Usually, the breaks in a parsed string occur at the spaces. For example:

```
PARSE VALUE "Five golden rings" WITH var1 var2 var3
```

This instruction assigns:

```
var1 the string "Five".
var2 the string "golden".
var3 the string "rings".
```

If you want to refer to the specific character position in the input string where you want the parsing to break, specify the number of characters from the beginning of the string. That is, the first character is 1, the second is 2, and so forth.

```
PARSE VALUE "Five golden rings" WITH var1 10 var2 15 var3
```

This *template* assigns:

```
var1 the string "Five gold" (up to the 9th position)
var2 the string "en ri"     (positions 10 to 14)
var3 the string "ngs"       (position 15 to the end)
```

Figure 8-12 shows an example that uses PARSE VALUE to parse the string returned by REXX's built-in TIME() function. The TIME() string has two digits each for hours, minutes, and seconds that are separated by colons. For example, 12:34:55.

```
/* parsing TIME() by absolute character position */
PARSE VALUE time() WITH hr 3 . 4 mn 6 . 7 sc .
say 'Hours   :' hr
say 'Minutes :' mn
say 'Seconds :' sc
```

Figure 8-12. ABSPTRN.CMD

Relative Position

You can also specify breaks by *relative* position; that is, by a given number of characters from the last break. To parse this way, use signed numbers, which are numbers preceded by plus or minus signs to indicate the direction to move the break. For example:

```
PARSE VALUE "Five golden rings" WITH var1 10 -5 var2 +8 var3 17
```

This instruction assigns:

```
var1 the string "Five gold" (up to the 10th position)
```

```
var2 the string "golden"    (back 5, forward 8; note the leading space)
var3 the string "ring"      (stopped at position 16)
```

Figure 8-13 shows an example that uses PARSE VALUE as applied to the TIME() string.

```
/* parsing TIME() by relative character position */
PARSE VALUE time() WITH hr 3 +1 mn 6 +1 sc .
say 'Hours   :' hr
say 'Minutes :' mn
say 'Seconds :' sc
```

Figure 8-13. RELPTRN.CMD

Though PARSE VALUE was used for these examples, you can use positional patterns with any PARSE instruction.

Variables in Patterns

In place of a literal string or a position, you can use a variable containing the string, the absolute position, or the relative position. To do this, enclose the variable name in parentheses, as in Figure 8-14.

```
char = ':'
parse value time() with hr (char) mn (char) sc .
say 'Hours   :' hr
say 'Minutes :' mn
say 'Seconds :' sc
```

Figure 8-14. VARPTRN1.CMD

In fact, you can use a variable assigned in the same template, as in Figure 8-15.

```
parse value time() with hr 3 char +1 mn (char) sc .
say 'Hours   :' hr
say 'Minutes :' mn
say 'Seconds :' sc
```

Figure 8-15. VARPTRN2.CMD

To Summarize

Any data can be parsed using patterns in the template of a PULL, ARG, or PARSE instruction. A template is recognized as a pattern where there is:

- A literal string, such as / in the example TAKE.CMD (see Figure 8-11 on page 8-10)
- A symbol in parentheses, which means that it is the name of a variable
- An unsigned number, which means that parsing continues at the specified character position
- A signed number, which means that parsing continues at the specified character position, relative to the first character of the last match.

String Functions

Another kind of parsing can be performed using REXX's built-in string functions. They are grouped here by the tasks they perform. For more information, refer to "Functions" in the *REXX Reference.*

You can use the following functions to separate and change input strings.

Getting Pieces

Substring functions that get (return a piece of a larger string) are:

SUBSTR()	Gets a piece of a string by numbered position
LEFT()	Gets the leftmost substring; can add trailing spaces
RIGHT()	Gets the rightmost substring; can add leading spaces
WORD()	Gets a word from a string (by number)
SUBWORD()	Gets a substring beginning with a given word.

Editing

Functions that change a string are:

INSERT()	Inserts a substring into a string
OVERLAY()	Overlays part of one string with another
REVERSE()	Swaps the characters in a string, end for end
COPIES()	Replicates a string a given number of times.

Deleting

Functions that delete substrings are:

DELSTR()	Deletes a substring from the input string
DELWORD()	Deletes a substring from the input string, beginning with a given word.

Formatting

Functions that change a string by adding or removing spaces or other characters are:

SPACE()	Adds or deletes intervening spaces (or other delimiting characters) between words
CENTER()	Centers the input string within a larger string of a given length; adds spaces (or other characters)
STRIP()	Removes leading or trailing spaces (or both) from a string.

Note: The previously described LEFT and RIGHT functions can also add spaces.

Counting

Functions that get the lengths of strings, compare strings, or locate a particular character position within a string are:

LENGTH()	Counts the characters in a string
WORDS()	Counts the words in a string
WORDLENGTH()	Returns the length of a word (specified by number).

Comparing

Functions that return a number, based on a comparison of two strings are:

VERIFY() Determines whether one string is made up of characters in another. It can return the position of either the first matching character or the first nonmatching character (the default).

ABBREV() Returns 1 (true) if one string matches the leading characters of another.

COMPARE() Determines if two strings are identical.

Finding Positions

Functions that return a number that is a sought-after character position are:

POS() Searches one string, from the beginning, for the presence of a given substring and returns the substring's position

LASTPOS() Searches one string, from the end, backward, for the presence of a given substring and returns the substring's position

WORDINDEX() Searches for a word by number and returns the initial character position

WORDPOS() Seaches for a word by the word itself and returns the initial character position.

Examples

Figure 8-16 shows an example of a REXX program that may be called as a string function. It changes an input number into a string in currency format. It uses PARSE VAR with a literal pattern and PARSE VALUE with a character-position pattern.

```
/* Takes a number and returns a string in comma-delimited          */
/* dollar format; e.g., DOLLAR(1234.5555)  returns '$1,234.56'     */

arg number                              /* get the argument NUMBER  */

/* Round off the argument to the nearest cent, then      */
/* parse the result into the integer (DOLLARS), the      */
/* decimal point and the decimal fraction (CENTS)        */
/* (NOTE: More about the FORMAT() function on page 9-6.) */

PARSE VALUE format(number,,2,0) WITH dollars "." cents

dollars = abs(dollars)                  /* make DOLLARS positive    */

backin = reverse(dollars)               /* reverse the digits in    */
                                        /* DOLLARS so we can parse  */
                                        /* them into groups of 3    */
                                        /* (see REVERSE(), above)   */

backout = ""                            /* initialize a variable    */
                                        /* for re-concatenation     */

do while length(backin) > 3             /* while three digits or    */
                                        /* more remain in BACKIN,   */
   PARSE VAR backin group 4 backin      /* take each group of three */
                                        /* remaining digits, and    */
   backout = backout||group||","        /* then join it to the end  */
                                        /* of the BACKOUT variable  */
                                        /* and add a comma          */
   end

backout = backout||backin||"$"          /* concatenate the digits   */
                                        /* that remain; add '$'     */

if number < 0 then                      /* if the argument was      */
   backout = backout||"-"               /* negative, restore the    */
                                        /* minus sign               */

bucks = reverse(backout)||"."||cents    /* restore the proper order */
                                        /* of the digits; add the   */
                                        /* decimal point and cents  */

return bucks                            /* return the string        */
```

Figure 8-16. DOLLAR.CMD

Figure 8-17 shows an example of how DOLLAR.CMD could be used as a function in the program SUM.CMD (see Figure 6-10 on page 6-13).

```
/* using the function DOLLAR() in a program */
total = 0
do forever
   say "Type amount:"
   pull entry
   if \datatype(entry,n)           /*if entry is not a valid number  */
   then leave                       /* leave the loop                 */
   total = total + entry
   say "Total = " DOLLAR(total)     /* display TOTAL in dollar format */
end
say entry "is not a number.  Returning to OS/2."
```

Figure 8-17. SUMCASH.CMD

Figure 8-18 shows an example of a useful search-and-replace string function. Notice how the second PARSE instruction uses a variable as a pattern.

```
/* Function:  CHANGE(string,old,new)                    */
/* Changes all occurrences of "old" in "string"         */
/* to "new".  If "old" == "", then "new" is prefixed    */
/* to "string".                                         */

parse arg string, old, new
if old=="" then return new||string

out=""
do while pos(old,string)\= 0
   PARSE VAR string prepart (old) string
   out=out||prepart||new
end
return out||string
```

Figure 8-18. CHANGE.CMD

Figure 8-19 shows an example using the CHANGE() function.

```
/* using the CHANGE() function  */
direction = "north by northwest"
wrong = "north"
right = "south"
say direction
say change(direction,wrong,right)   /* displays "south by southwest" */
```

Figure 8-19. CHNGDEMO.CMD

Chapter 9. Arithmetic

This chapter discusses using REXX for calculating and displaying numbers. For more detailed information, refer to the *REXX Reference*.

Basics

In this chapter:

Basics

- About REXX numbers
- Checking input numbers
- Calculating
- Formatting output.

About REXX Numbers

Character strings on which REXX can perform arithmetic operations are referred to as *numbers*.

In REXX, a number is a string of digits (0 through 9). A number must begin with a digit, a plus sign (+), or a minus sign (−). A single decimal point is permitted. The letter E (or e) can be used to denote powers of 10.

The NUMERIC DIGITS instruction controls the number of decimal places allowed. The default is nine places.

REXX ignores leading and trailing spaces in number strings. It does not allow spaces or commas within a number.

These are some examples of valid REXX numbers:

12 This is a *whole number* or *integer*.

−5 This is a *signed* number (minus five).

0.5 This is a *decimal fraction* or *decimal* (one half).

3.5E6 This is a *floating point* number (three and one-half million). It uses *exponential notation*. This notation is useful when dealing with very large or very small numbers. The portion that follows the E is the number of places the decimal point must be moved to the right to make it into an ordinary number.

Checking Input Numbers

Before a program tries to do arithmetic on data typed from the keyboard, the data should be checked to verify that it has valid numbers to work with. You can do this by using the DATATYPE() function. For example:

```
datatype(expression,[type])
```

where

`expression` is the value to be tested.

`type` is an optional argument, the kind of data you are testing for.

In its simplest form, DATATYPE() returns the string NUM if the argument (the expression inside the parentheses) is accepted by REXX as a number that could be used in arithmetical operations. Otherwise, it returns the string CHAR.

The value of	*is the string*
datatype(49)	"NUM"
datatype(5.5)	"NUM"
datatype(5.5.5)	"CHAR"
datatype("5,000")	"CHAR"
datatype(5 4 3 2)	"CHAR"

Figure 9-1 shows an example that requires the user to keep trying until a valid number is typed.

```
/* Example requiring numeric input */
do until datatype(howmuch) = "NUM"
   say "Type a number"
   pull howmuch
   if datatype(howmuch) = "CHAR" then
     say "That was not a number.  Try again!"
end

say "The number you typed was" howmuch
```

Figure 9-1. VALNUM.CMD

To test for a particular type of data, such as whole numbers, use the alternative form of the DATATYPE() function. This form requires two arguments:

- The expression to be tested
- The type of data to be tested; for example, whole for a whole number.

  Only the first character of type is inspected. To test for whole numbers, it would be sufficient to write W or w. In this book, *whole* is used to remind you of the meaning of this argument.

This form of the function, DATATYPE(number,whole), returns 1 (true) if number is a whole number, 0 (false) if otherwise. For example:

```
do until datatype(howmany,whole)

⋮
   pull howmany

⋮
end
```

If you also want to restrict the input to numbers greater than 0, you could write:

```
do until datatype(howmany,whole) & howmany > 0

⋮
   pull howmany

⋮
end
```

The ampersand (&) is an operator that combines conditions (see "Combining Expressions" on page 4-6). In this example, both `datatype (howmany,whole)` *and* `howmany > 0` must be true for the loop to end.

The DATATYPE() function can test for other types of data, as well. For examples, see the *REXX Reference*.

Calculating

Addition and Subtraction

These operations are performed in the usual way. You can use both whole numbers and decimal fractions.

Operator	Operation	Example
+ (plus sign)	Add	`say 7 + 2  /* displays '9' */`
− (minus sign)	Subtract	`say 7 − 2  /* displays '5' */`

Multiplication and Powers

Operator	Operation	Example
* (asterisk)	Multiply	`say 7 * 2  /* displays '14' */`
** (double asterisk)	Raise to a whole number power	`say 7**2  /* displays '49' */`

Division

When you divide, you determine whether or not you want the answer expressed as a whole number (integer). The operators you can use are:

/ (one slash) — Divide. For example:

```
say 7 / 2   /* displays '3.5' */
```

% (percent sign) — Integer divide. The result is a whole number. Any remainder is ignored. For example:

```
say 7 % 2   /* displays '3'   */
```

// (two slashes) — Remainder after integer division. For example:

```
say 7 // 2  /* displays '1'   */
```

Notice which of these operators is used in the example shown in Figure 9-2.

```
/* This program works out how to share zero or more     */
/* sweets between one or more children, assuming that   */
/* a single sweet cannot be split.                      */

/*------------------------------------------------------*/
/* Get input from user                                  */
/*------------------------------------------------------*/
do until datatype(sweets,whole) & sweets >= 0
   say "How many sweets?"
   pull sweets
end

do until datatype(children,whole) & children > 0
   say "How many children?"
   pull children
end

/*------------------------------------------------------*/
/* Compute result                                       */
/*------------------------------------------------------*/
say "Each child will get" sweets%children  "sweets",
    "and there will be"   sweets//children  "left over."
```

Figure 9-2. SHARE.CMD

Do not to try to divide by 0. If you do, a syntax error results. That is why, in Figure 9-2, the user was not allowed to answer 0 to the question "How many children?".

Because apples and oranges can be cut into pieces, you can use the / division operator. For example:

```
children = 5;  apples = 7;
say "Each child gets" apples/children "apples."

/* displays 'Each child gets 1.4 apples.' */
```

Fractions are usually computed with an accuracy of nine significant digits. For example:

```
children = 3;  oranges = 7;
say "Each child gets" oranges/children "oranges."

/* displays 'Each child gets 2.33333333 oranges.' */
```

To summarize:

- The result of a % operation is always a whole number, the quotient only.
- There may be a remainder. To compute the remainder, write the expression using the // operator.
- The result of a / operation can be a decimal.

Range and Precision

REXX calculates the result correct to nine digits if necessary. This means nine significant digits, not counting the zeros that come immediately after the decimal point in very small decimal fractions. For example:

```
say 1*2*3*4*5*6*7*8*9*10*11*12      /* displays '479001600' */

say 7/30000000000       /* displays '0.000000000233333333' */
```

The accuracy of computed results can be changed using the NUMERIC DIGITS instruction. See "Changing Precision" on page 9-16.

Exponential Notation

Numbers much larger or smaller than those previously discussed are difficult to read and write, because it is easy to make a mistake counting the zeros. It is simpler to use *exponential notation*. Very large numbers can be written as an ordinary number, followed by a letter E, followed by a whole number, called the *exponent*. The exponent is how many places to the right (positive exponent) or left (negative exponent) that the decimal point of the fixed-point number would have to be moved to obtain the same value as an ordinary number. For example:

4.5E6 is the same as 4 500 000 (four and one-half million)
23E6 is the same as 23 000 000 (twenty-three million)
1E12 is the same as 1 000 000 000 000 (a million million)
4.5E−3 is the same as 0.004 5 (four and one-half thousandths)
1E−6 is the same as 0.000 001 (one millionth).

Write numbers this way in expressions and also when entering numeric data requested by REXX programs. REXX uses this notation when displaying results that are too large or too small to be expressed conveniently as ordinary numbers or decimals. When REXX uses this notation, the part of the number that comes before the E (the *mantissa*) is usually a number between 1 and 9.999 999 99. For example:

```
j = 1
do until j > 1e12
   say j                  /* displays '1'              */
   j = j * 11             /*          '11'             */
end                       /*          '121'            */
                          /*          '1331'           */
                          /*          '14641'          */
                          /*          '161051'         */
                          /*          '1771561'        */
                          /*          '19487171'       */
                          /*          '214358881'      */
                          /*          '2.35794769E+9'  */
                          /*          '2.59374246E+10' */
                          /*          '2.85311671E+11' */
```

Numbers written in exponential notation, such as 1.5E9, are sometimes called *floating-point* numbers. Conversely, ordinary numbers, such as 3.14, are sometimes called *fixed-point* numbers.

Test Yourself

What is displayed on the screen when you run the program shown in Figure 9-3?

```
/* Example: arithmetical operations */
quarter = 25
deuce = 2
say quarter + deuce
say quarter - deuce
say quarter * deuce
say quarter / deuce
say quarter % deuce
say quarter // deuce
x = quarter"E"deuce
say x + 0
```

Figure 9-3. ARITHOPS.CMD

Answer: The following is displayed on the screen.

```
[C:\] arithops
27
23
50
12.5
12
1
2500
C:\[]
```

The last two lines of the program require some explanation. First, x gets the value 25E2. This is the same as 25.00 with the decimal point moved two places to the right (in other words, 2500). When x is used in the arithmetical expression, the number 25E2 is added to 0, giving a result of 2500.

Formatting Output

Your program has accepted numbers from its user and made calculations. Now it must display the results.

When formatting output, REXX:

- Rounds numbers automatically before every operation, if necessary. Rounding is to the current precision value set by a NUMERIC DIGITS instruction (see page 9-16) or to the default value of nine decimal digits.
- Removes all leading zeros from the integer part of number, but leaves in the trailing zeros in the decimal portion.

You may want to keep the precision of the calculations to the ninth place, but you do not need to display all those extra zeros. That is what the following two functions are for:

```
TRUNC(number,places)
```

TRUNC() returns just the integer part of number and the number of decimal places you specify. The only remaining decimals are truncated or *cut off*. That is, TRUNC *rounds down*. For example:

```
say trunc(321.765,2)  /* displays 321.76. */
```

Likewise, FORMAT() function leaves a specified number of decimals, but it does *conventional rounding*. The last decimal remaining is increased by one if the next (truncated) decimal is 5 or greater. For example:

```
FORMAT(number,before,after)
```

The first three arguments of the FORMAT() function are:

The number to be formatted
How many digits to be shown before the decimal point
How many digits to be shown after the decimal point.

For example:

```
say format (321.765,3,2)   /* displays 321.77 */
```

Note: 321.7649 rounded to two places is 321.76, because FORMAT() only uses the first truncated digit to determine rounding.

Figure 9-4 shows an extended example to compare FORMAT() and TRUNC().

```
/*  An example of rounding and displaying numbers */
say
say "       At full         Round w/   Round w/ "
say "       precision       FORMAT()   TRUNC()"
say copies("-",45)

do num = 1 to 9
   quotient = 23 / num

   norm = FORMAT(quotient,3,3) /* rounding normally to 3 decimal places */
   down = TRUNC(quotient,3)    /* rounding down to 3 decimal places     */

   /* Now, we will use FORMAT() to put the numbers into columns */

   say "23/"num "=" FORMAT(quotient,2,8) FORMAT(norm,6,3) FORMAT(down,6,3)

end
```

Figure 9-4. ROUNDING.CMD

The following is displayed on the screen, when you run ROUNDIT.CMD.

```
[C:\]roundit

        At full         Round w/    Round w/
        precision       FORMAT()    TRUNC()
-----------------------------------------------
23/1 = 23.00000000      23.000      23.000
23/2 = 11.50000000      11.500      11.500
23/3 =  7.66666667       7.667       7.666
23/4 =  5.75000000       5.750       5.750
23/5 =  4.60000000       4.600       4.600
23/6 =  3.83333333       3.833       3.833
23/7 =  3.28571429       3.286       3.285
23/8 =  2.87500000       2.875       2.875
23/9 =  2.55555556       2.556       2.555
[C:\]
```

Summary

This completes "Basics" in this chapter. You have learned how to:

- Check user input
- Make REXX do the calculating
- Display the results on the screen.

"Advanced Topics" in this chapter discusses:

- Additional formatting techniques
- How to control the precision of REXX arithmetic
- Comparing number strings
- Scientific notation and exponentiation.

To continue with "Basics," go to page 10-1.

Advanced Topics

In this chapter:

Advanced Topics

- Putting numbers into columns
- Formatting errors
- Rounding errors
- Conventional and scientific notation
- Changing precision
- Comparing numbers
- Powers (** operator)
- A square-root function.

Putting Numbers into Columns

Columns of figures are easier to read if the numbers are aligned with the units in the same column. The FORMAT() function helps you to do this (see Figure 9-5).

```
/* Example showing how columns of figures are formatted */

qty.1 = 101;      unitprice.1 = 0.73;      remark.1 = OK
qty.2 = 500;      unitprice.2 = 1995;      remark.2 = OK
qty.3 = 60000;    unitprice.3 = 70000;     remark.3 = OK
qty.4 = 500;      unitprice.4 = 400/12;    remark.4 = OK

say "Quantity    Unit Price    Total Price   Observations"

do item = 1 to 4
   say format(qty.item, 5,0),
       format(unitprice.item, 11,2),
       format(qty.item * unitprice.item, 12,2),
       "  " remark.item
end
```

Figure 9-5. INVOICE.CMD

The following formatted data is displayed on the screen.

```
Quantity    Unit Price    Total Price   Observations
  101          0.73          73.73    OK
  500       1995.00       997500.00    OK
60000      70000.00           4.20E+9    OK
  500         33.33        16666.67    OK
```

Formatting Errors

The numbers to be formatted should always be small enough to fit into the space you have reserved for them with FORMAT.

A simple rule is to always specify at least nine spaces for the *before the decimal point* argument. To ensure that numbers with more than nine digits are displayed in

exponential notation. The extra characters required causes fields to the right of the number to shift to the right, thus drawing attention to the exception.

Look at item 3 in the previous example. The quantity times the unit price (60 000 times 70 000) gives a total price of 4 200 000 000, which is too large for the nine-digit field that was specified. The result has therefore been displayed in exponential notation. This in turn has caused OK to be shifted right.

If we add `qty.5 = 880 000, unitprice.5 = 1,` and `remark.5 = "Big deal"` and change the 4 to a 5 in the DO instruction, then the following is displayed on the screen.

```
[C:\]invoice
Quantity    Unit Price    Total Price   Observations
    101          0.73          73.73    OK
    500       1995.00       997500.00    OK
  60000      70000.00           4.20E+9    OK
    500         33.33        16666.67    OK
     12 +++  say format(qty.item, 5,0),       format(unitprice.item, 11, 2),
 format(qty.item * unitprice.item, 12,2),       " " remark.item
REX0040: Error 40 running C:\INVOICE.CMD, line 12: Incorrect call to
routine

[C:\]
```

The error could have been avoided by:

- Testing the input values for a maximum number of 99 999

  or

- Allowing space enough for at least nine digits for the integer part. For example:

  ```
  say format(qty.item, 9,0),
      format(unitprice.item, 9,2),
      format(qty.item * unitprice.item, 11,2),
      " " remark.item
  ```

  The following formatted data is displayed on the screen.

```
Quantity    Unit Price    Total Price   Observations
     101          0.73          73.73    OK
     500       1995.00       997500.00    OK
   60000      70000.00           4.20E+9    OK
     500         33.33        16666.67    OK
  880000          1.00       880000.00    Big deal
[C:\]
```

Rounding Errors

When your program performs a series of arithmetical operations, additional errors can be inadvertently introduced. Look at the fourth item in the INVOICE.CMD program shown in Figure 9-5 on page 9-9. The customer seems to have been overcharged by $1.67. The price was $400 a dozen. FORMAT() has rounded this to 33.33 each. But `Total Price` was not rounded until after it had been multiplied by 500.

For rounding numbers, use FORMAT() at the point in your calculations where you want rounding to occur. For rounding down, use TRUNC().

Test Yourself

Write a program that accepts input liquid quantities and unit prices and displays the price per gallon and per liter.

Answer: Figure 9-6 shows an example, divided into six parts, of one way to do this.

```
/* Calculate a conversion table for input liquid quantities          */

/*   Simple variables (input and program control):                   */
/*   INPUTQTY  : Quantity                                            */
/*   INPUTUPR  : Unit price                                          */
/*   UNIT      : Unit of measure (G or L)                            */
/*   RPTUPR    : Unit price of previous entry                        */
/*   RPTUNIT   : Unit of measurement of previous entry               */
/*   ITEM      : Item number (each purchase)                         */

/*   Compound symbols (for each item of output):                     */
/*   QTYINGAL.ITEM  : Quantity in gallons                            */
/*   UPPGAL.ITEM    : Unit price per gallon                          */
/*   QTYINLIT.ITEM  : Quantity in liters                             */
/*   UPPLIT.ITEM    : Unit price per liter                           */
/*   TTLPRC.ITEM    : Total price of purchase                        */

/* MAIN PROGRAM: Calls input subroutines GETQTY, GETUPR, and GETUNIT */
/* and CALCULATE subroutine ; results are then displayed in table.   */

inputqty = ""              /* Initialize variable for quantity input */
                           /* (also controls end of input).          */

do item = 1                /* For each item...                       */

   call getqty             /* - get input for quantity               */
   if inputqty = "" then   /*   if none entered, then quit the loop  */
      leave                /*   and display results                  */

   call getupr             /* - get input for unit price             */
   call getunit            /* - get unit of measurement for input    */
   call calculate          /* - do the calculations                  */
   inputqty = ""           /* re-initialize input variable           */

end                        /* and repeat the loop                    */
```

Figure 9-6 (Part 1 of 6). LIQUID.CMD

```
/* When all data have been input, display a header ....                */
say
say "    ¦    IN LITERS      ¦    IN GALLONS     ¦     TOTAL"
say "    ¦ quantity    unit  ¦ quantity    unit  ¦     PRICE"
say "    ¦            price  ¦            price  ¦          "
say copies("=",56)
/* ... then display the calculated values in columns using FORMAT().  */
last = item -1
do item = 1 to last
   say,
        format(item,3,0)" ¦",                    /* Item number      */
        format(qtyinlit.item,5,2),               /* Qty. in liters   */
        format(upplit.item,3,3)"  ¦",            /* Price per liter  */
        format(qtyingal.item,5,2),               /* Qty. in gallons  */
        format(uppgal.item,3,3)"  ¦",            /* Price per gallon */
        format(ttlprc.item,7,2)                  /* Total purchase   */
end
exit
```

Figure 9-6 (Part 2 of 6). LIQUID.CMD

```
/*     Following are the subroutines for data input                    */

getqty:           /* Accept input quantity for each item.              */
/* The DO loop allows the user to type only a number or (by           */
/* pressing the Enter key alone) a null string, ending the input loop.*/

do until datatype(inputqty,n) ¦ inputqty = ""
   say
   say "Type quantity for item number" item
   say "  or press the Enter key alone to end the program."
   pull inputqty
end
return
```

Figure 9-6 (Part 3 of 6). LIQUID.CMD

```
getupr:          /* Accept input for each item's unit price.          */
/* The SYMBOL() function tests whether previous input exists          */
/* (i.e., if the variable REPTUPR has been assigned a value).         */
/* If so, the user can 'carry over' that previous input by            */
/* simply pressing the Enter key.                                     */

say
say "Type unit price"
if symbol("reptupr") <> "LIT" then
   say "  (or press the Enter key for" reptupr")"

/* Again, the DO loop allows only numeric input.                      */

do until datatype(inputupr,n)
   pull inputupr             /* Entering a null string (i.e.,        */
   if inputupr = "" then     /* pressing the Enter key only)         */
      inputupr = reptupr     /* 'carries over' the previous item's   */
   else reptupr  = inputupr  /* unit price; otherwise this input is  */
end                          /* stored for later 'carry-over.'       */
return
```

Figure 9-6 (Part 4 of 6). LIQUID.CMD

```
getunit:         /* Accept input for measure (liters or gallons);    */
/* again, the SYMBOL function checks for prior use of repeat variable */

say "Type G for gallons or L for liters"
if symbol("reptunit") <> "LIT" then
   say "  (or press the Enter key for" reptunit")"

/* The DO loop allows only a "G" or "L" as input                      */

do until unit = "G" ¦ unit = "L"
   pull unit
   if unit = "" then       /* A null string is changed to           */
      unit = reptunit      /* the previous input; otherwise         */
   else reptunit = unit    /* this entry is stored for next time.   */
end
return
```

Figure 9-6 (Part 5 of 6). LIQUID.CMD

```
calculate:       /* Calculate the output values according to unit     */

select
   when unit ="G" then do                 /* if input is in gallons...  */
     qtyingal.item = inputqty             /* store the input quantity   */
     qtyinlit.item = inputqty*3.7853      /* convert to liters          */
     uppgal.item  = inputupr              /* store the unit price       */
     upplit.item  = inputupr/3.7853       /* compute 'per liter' price  */
     end
   when unit ="L" then do                 /* if input is in liters...   */
     qtyinlit.item = inputqty             /* store the input quantity   */
     qtyingal.item = inputqty/3.7853      /* convert to gallons         */
     upplit.item  = inputupr              /* store the unit price       */
     uppgal.item  = inputupr*3.7853       /* compute 'per gallon' price */
     end
end

ttlprc.item = inputqty * inputupr        /* compute total price of item */
return
```

Figure 9-6 (Part 6 of 6). LIQUID.CMD

Conventional and Scientific Notation

You can control whether numbers are expressed in conventional or scientific notation with the FORMAT() function.

Fixed-point (Conventional) Notation

To stop FORMAT() from returning floating-point numbers (when results would usually be expressed in floating-point numbers), use the fourth argument of FORMAT. This argument specifies the number of character positions reserved for the exponent. Exponential notation is not used if you write:

```
FORMAT(number,before,after,0)
```

Be sure that the amount of space you have allowed for `before` and `after` is sufficient.

Floating-point (Scientific) Notation

To make FORMAT() return floating-point numbers (called exponential or scientific notation) when the results would usually be expressed in fixed-point numbers, use the fifth argument of FORMAT(). This argument specifies the *threshold* for expressing the result in exponential notation. Exponential notation is used if you write:

```
FORMAT(number,before,after,,0)
```

Note: When a floating-point number has an absolute value between 1 and 9.999 999 99 (that is, when the **exponent** is 0) the characters `E+0` are always omitted, even when floating-point has been specified.

For other uses of the FORMAT() function, see the *REXX Reference*.

Test Yourself

1. Write a program called REFORMAT that expresses numbers typed by the user in both fixed-point and exponential notation.
2. Test your program with the numbers:

   123 456 789
   0.000 000 000 001 234 5
   999 999 999 999e-6
   1.2e10
   1.2
   1.2e+0

Answers:

1. Figure 9-7 shows an example of a possible answer.

```
/* Example: to change the format of a number */
do forever
   say "Type a number"
   pull answer
   if \ datatype(answer,number) then exit
   say "Fixed-point equivalent:" format(answer,,,0)
   say "Exponential equivalent:" format(answer,,,,0)
end
```

Figure 9-7. REFORMAT.CMD

2. Figure 9-8 shows a table that lists the results you should get when using the test numbers with the REFORMAT.CMD.

Number typed	Fixed point equivalent	Exponential equivalent
123 456 789	123 456 789	1.234 567 89E+8
0.000 000 000 001 234 5	0.000 000 000 001 234 5	1.2345E-12
999 999 999 999e-6	1 000 000.00	1.000 000 00E+6
1.2e10	1 200 000 000 000	1.2E+10
1.2	1.2	1.2
1.2e+0	1.2	1.2

Figure 9-8. REFORMAT.CMD Results

Changing Precision

If you want to avoid using exponential notation, or if you want to control the precision of your calculations, use the NUMERIC DIGITS instruction to change the number of significant digits (see Figure 9-9). (The default setting for NUMERIC DIGITS is 9.)

```
/* examples of numbers with unusually high precision */

numeric digits 10
say "The largest signed number that can be held"
say "in a C long integer is" 2**31 - 1 "exactly."
say

numeric digits 48
say "1/7 =" 1/7
```

Figure 9-9. ACCURATE.CMD

The following is displayed on the screen when you run the program.

```
[C:\] accurate
The largest signed number that can be held
in a C long integer is 2147483647 exactly.

1/7 = 0.142857142857142857142857142857142857142857142857
[C:\]
```

To check the current setting of the NUMERIC DIGITS instruction, use the DIGITS() function, `digits()`. If you have not used the NUMERIC DIGITS instruction to change precision, then the DIGITS() function returns 9 (the default setting).

Rounding and Precision

The NUMERIC DIGITS instruction specifies the maximum number of significant digits in the result of a REXX calculation. Whatever the setting you give for NUMERIC DIGITS, REXX carries out its calculations to an even higher precision. Extra *guard* digits are provided for each input number. Multiplication and division are calculated out to twice the NUMERIC DIGITS setting.

This means that the only arithmetic errors that can occur are in the final rounding. For example:

```
numeric digits 3
say 100.3 + 100.3          /* displays 201                */
                           /* 200.6 is rounded to '201'   */
```

Unless you have a specific need for high-precision calculations, leave NUMERIC DIGITS at the default setting of 9. Higher settings can slow your programs down needlessly. Lower settings can affect calculations in a way you do not intend (for example, how DO loop counters are incremented). When you need to round final results for printouts or displays, use the FORMAT() function.

For more information about rounding, see "Numerics and Arithmetic" in the *REXX Reference*.

Comparing Numbers

There are times when an accurate comparison is inconvenient (see Figure 9-10).

```
/* Example: no approximation here */

say 1 + 1/3                    /* displays '1.33333333'   */
say 1 + 1/3 + 1/3 + 1/3        /* displays '1.99999999'   */
say 1 + 1/3 + 1/3 + 1/3 = 2    /* displays '0'  (false)   */
```

Figure 9-10. NOFUZZ.CMD

To make REXX comparisons allow for approximate values (make them less accurate than ordinary REXX arithmetic), use the instruction, `NUMERIC FUZZ n`, where `n` is the number of digits (at full precision) that REXX should ignore when comparing numbers. The number `n` must be a whole positive number (or any expression that is so evaluated), and it must be less than the current NUMERIC DIGITS setting (see Figure 9-11).

```
/* Example: allowing approximation */
say 1 + 1/3 + 1/3 + 1/3 = 2    /* displays '0'  (false)  */
numeric fuzz 1
say 1 + 1/3 + 1/3 + 1/3 = 2    /* displays '1'  (true)   */
```

Figure 9-11. FUZZ.CMD

To check the current setting of the NUMERIC FUZZ instruction, use the function, `FUZZ()`. If no NUMERIC FUZZ setting has been given, FUZZ() returns `0` by default. This means that the 0 digits are ignored during a comparison operation.

Powers (** Operator)

The operator ** means *raised to the whole-number power of.* For example:

```
2**1 = 2         =  2  (2 to the power of 1)
2**2 = 2*2       =  4  (2 to the power of 2, or 2 squared)
2**3 = 2*2*2     =  8  (2 to the power of 3, or 2 cubed)
2**4 = 2*2*2*2   = 16  (2 to the power of 4).
```

As in ordinary algebra:

```
2**0   = 1
2**-1  = 1/(2**1) = 0.5  (2 to the power of minus 1)
2**-2  = 1/(2**2) = 0.25 (2 to the power of minus 2).
```

Note: The number on the right of the ** *must* be a whole number.

In the order of precedence built into REXX, the powers (**) operator comes below the *prefix operators* and above the multiply and divide operators. For example:

```
say -5**2        /* displays '25'.  Same as (-5)**2         */
say 10**3/2**2   /* displays '250'.  Same as (10**3)/(2**2) */
```

Test Yourself

1. Examine the program shown in Figure 9-12.

```
/* Example of a negative exponent */
if 2 ** -3 = 1/(2**3) then say "True"
else say "False"
```

Figure 9-12. EXPONENT.CMD

   a. What is displayed on the screen when you run this program?
   b. Are the parentheses in this expression really necessary?

2. What value is computed for the expression:

```
say      9 ** (1/2)
```

Answers

1. In EXPONENT.CMD

   a. The word True is displayed on the screen.
   b. No. The ** operator has a higher priority than the / operator, so REXX evaluates the expression in the same way if the parentheses were removed.

2. The result is a syntax error. The decimal form (9 ** 0.5) does not work either. True, in mathematics, *x to the power of 1/2* means *the square root of x*. But in REXX, the ** operator *must* be followed by a whole number or by an expression that, when evaluated, gives a whole number.

A Square-Root Function

Figure 9-13 shows an example of a SQRT() function written as a REXX program.

```
/* The SQUARE ROOT function                                    */
/*                                                             */
/*                SQRT(number)                                 */
/*                                                             */
/* where "number" is a non-negative REXX number,               */
/* returns the square root of "number".  Precision is          */
/* nine significant figures, independent of the setting        */
/* of NUMERIC DIGITS (explained on page 9-16).                 */
/*                                                             */
/* Implementation details:  "number" is normalized to          */
/* give an even exponent (so that the exponent can be          */
/* dealt with separately later) and a mantissa between         */
/* 1 and 100.  The most significant digit of the result        */
/* is found.                                                   */
/*                                                             */
/* The mantissa is multiplied by 100, the exponent is          */
/* reduced by 2 to compensate, the partial result              */
/* (ROOT) is multiplied by 10 (leaving a 0 in the              */
/* units position) and the units digit of this partial         */
/* result is then found, and so on.                            */
/*                                                             */
/* Finally, the result is adjusted using the computed          */
/* exponent.                                                   */

/*-------------------------------------------------------------*/
/* Set precision                                               */
/*-------------------------------------------------------------*/
numeric digits 10             /* for partial results           */
                              /* use one digit more            */
                              /* than final precision          */
/*-------------------------------------------------------------*/
/* Check arguments                                             */
/*-------------------------------------------------------------*/
if arg() \= 1
   then return                /* wrong number arguments        */
arg x
if \ datatype(x,number)
   then return                /* argument not a number         */
if x < 0
   then return                /* argument is negative          */

                                    /* continued ...           */
```

Figure 9-13 (Part 1 of 2). SQRT.CMD

```
/*-------------------------------------------------------*/
/* Normalize:                                            */
/* FROM                                                  */
/*    x      the argument                                */
/* COMPUTE                                               */
/*    mant   the mantissa, where 0 < mant < 100          */
/*    exp    the exponent, where exp is even             */
/*-------------------------------------------------------*/
                        /* Format so that 0 < mant < 10 */
parse value format(x,,,,0) with  mant "E" exp

                        /* Modify so that exp is even   */
if exp = "" then exp = 0
if exp//2 \= 0 then do
   mant = mant * 10
   exp = exp - 1
end

/*-------------------------------------------------------*/
/* Find root by successive approximation                 */
/*-------------------------------------------------------*/
root = 0
do 10
  do digit = 9 by -1 to 0,      /* find largest digit,  */
     while,                     /* such that            */
       (root + digit)**2 > mant /* (root+digit)squared  */
  end                           /*          is \>  mant*/
  root = root + digit
  if root**2 = mant then leave
  root = root * 10
  mant = mant * 100
  exp = exp - 2
end

/*-------------------------------------------------------*/
/* Adjust for computed exponent                          */
/*-------------------------------------------------------*/
numeric digits 9
return root * 10**(exp/2)
```

Figure 9-13 (Part 2 of 2). SQRT.CMD

Chapter 10. Input and Output

REXX can do more than manipulate the information that the user has typed at the keyboard and then process it for display on the screen. REXX can store, access, print, and organize data outside the program.

Basics

In this chapter:

Basics

- A stream of information
- Text file processing
- Writing data to a file
- Reading data from a file
- Printing a text file.

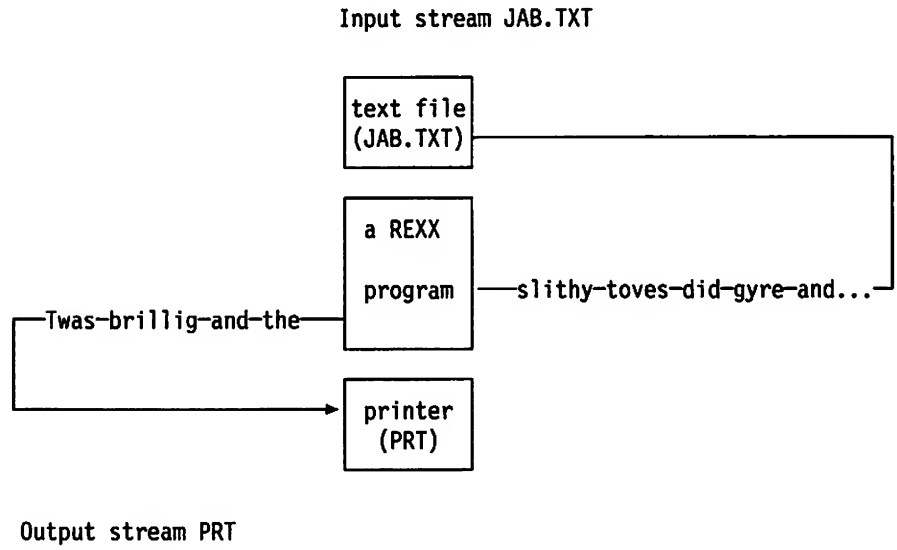
A Stream of Information

In computing, the form in which information comes is often as important as its content. A spreadsheet file, for example, contains not only the numbers and formulas put into it, but a good deal of other information that you never see, such as information about the structure of the file. This additional data describes how the file is organized and how it is stored and displayed.

The structure of a spreadsheet file is very different from that of a document formatted by a word processor. Information takes on other forms when it moves among the various *devices* in a system: the keyboard, the display, the printer, and so on.

The goal of the REXX language is to keep things as simple as possible. Therefore, REXX takes the simplest possible view of the information it receives. The simplest way to look at information is one character at a time. REXX sees external information as a *stream*, a long, single-file row of characters.

For example, in a REXX program that reads from a text file and then sends the text to a printer, both the file and the device to which the printer is connected are character streams:

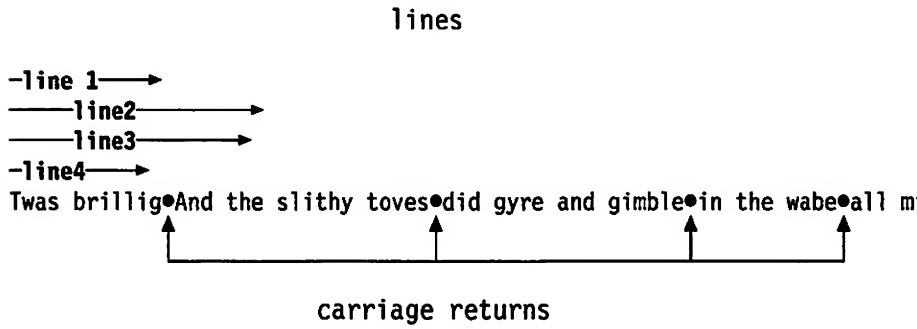


In the preceding diagram:

- The text file being read (JAB.TXT) is the *input stream.*
- The printer device written to (PRT) is the *output stream.*

In this discussion, a *stream* means any source or destination of external information used by a REXX program. A stream can be a disk file, the input from the keyboard or a data port, or the output to a printer or display.

In certain instances, as with text files, REXX can work with a stream in entire lines of data; that is, as strings of characters separated by carriage returns. For example:



REXX can arrange these lines into ordered lists, called *queues.*

The input and output operations of REXX fall into these broad categories:

- Streams of characters
- Lines, or segments of a stream, separated by carriage returns
- Queues, which are ordered lists of lines.

The way you use data streams in a program depends upon the kind of information you are working with and what you want to do with it.

Text File Processing

You begin text file processing by putting line data into more or less permanent form, such as in simple text files on disk. You already know some ways to create, read, and write disk files, either through application programs, such as your text editor, or through the OS/2 program.

In REXX, file processing is performed with a set of stream functions that read and write data a single character at a time or line-by-line. Because text files are usually organized into lines, the first function you try is one that writes in lines.

LINEOUT() Function

To write a line of text to a file, use the LINEOUT() function. For example:

```
lineout(stream,linedata)
```

where:

`stream` is name of the file (that REXX regards as a stream) to which the text is written.

`linedata` is a line of text (or any data) to be written.

The first time a program uses LINEOUT() in this way, the named `stream` is opened for writing and the line `linedata` is written to the end of the stream.

The stream remains open, and each subsequent LINEOUT() call writes a new line to the end of `stream`.

When the program ends, REXX automatically closes `stream`. Or, you can close `stream` explicitly at any time by omitting the `linedata` argument. For example:

```
lineout(stream)
```

Calling LINEOUT()

LINEOUT() is a function call rather than a keyword instruction. That means that it not only performs a given task (writing data to a file, in this example), but that it also *returns a value of*:

- `0` if `linedata` was successfully written to `stream`.
- `1` if for any reason `linedata` could not be written. For example, if you try to write to a *read-only* file.

The return value can be used by your program to detect whether something has gone wrong in the course of writing to a stream.

Note: If you use LINEOUT() without the `linedata` argument, the return value tells you if `stream` was successfully closed.

LINEOUT() is a function call; therefore, you have a choice about how you can use it. You can call LINEOUT:

- As part of a REXX instruction. For example, with the keyword SAY:

  ```
  say lineout("mybook.txt","Chapter 1.")  /* displays "0" if successful */
  ```

  or as a variable assignment:

  ```
  ready = lineout("mybook.txt","Chapter 1.") /* assigns "0" to READY   */
                              /* if successful          */
  ```

- As a subroutine with arguments (this is true of all function calls). For example:

```
call lineout "mybook.txt", "Chapter 1."
```

When you use LINEOUT() (or any function call) in this way, the return value of the function is automatically assigned to the REXX variable RESULT. For more information about calling functions in this way, see "Using a Call of the Other Kind" on page 7-13.

Writing Data to a File

Figure 10-1 shows an example of a simple text editor. It only writes new text to a file. Look closely at how the LINEOUT() function is used.

```
/* World's smallest editor */
say "Type file name"
pull filename
say "Type as many lines as you like.",
    "To finish, press the Enter key only."
do forever
   parse pull line


   if line = "" then           /* empty line?          */
      do
         call lineout filename /* if so, close the file */
         exit                   /* and end the program   */
         end


   /* otherwise, write LINE to the end of the file */
   call lineout filename, line
   if result = 1 then leave
end
say "Error, return code" rc
exit
```

Figure 10-1. EDDY.CMD

The user types a file name, which is then parsed by the PULL instructions and stored in the variable `filename`. As each line is typed:

- The PARSE PULL instruction stores it as a string in the variable `line`.
- The LINEOUT() function (called as a subroutine) writes the string contained in `line` to the file name stored in the variable `filename`.
- The DO loop continues until the user presses the Enter key twice, thereby entering a null string.
- The program then calls LINEOUT() with only the filename (closing the file), and then exits.

Reading Data from a File

To read the file and display the data on the screen, you could use the OS/2 command TYPE. Use a REXX program to add some extra features.

LINEIN() Function

To read a line from a stream into a REXX program, use the LINEIN() function. For example:

```
linein(stream)
```

where:

`stream` is name of the data stream (such as a file) from which the line is read.

The first time LINEIN() is called in a program, it opens the named `stream` and returns the first line of data. The second time it is called, it reads the second line and returns the second line of data, and so on until the program ends (or you close the stream with LINEOUT()). In other words, LINEIN() keeps track of its place in the stream with a kind of *bookmark*, called the *read position*. LINEOUT() uses a similar marker, called the *write position*. For more information, see "Accessing Data within a Stream" on page 10-20.

LINEIN() has no way of knowing when there are no more lines to read in a stream, such as when it gets to the end of a text file. To know when the read position has reached the end of the file, use the LINES() function.

LINES() Function

To find out if any lines remain between the read position and the end of a stream, the function, `lines(stream)`, returns:

`0` if no complete lines remain to be read

`1` if any lines remain.

Figure 10-2 shows an example, where stream is a text file , so LINES() would return 0 when the end of the file has been reached.

```
/* Program to display an entire file */
/* exits when end-of-file is reached */

say "Type a file name"          /* get the name of the file          */
pull filename                    /* from the user                     */

lineno = 1                       /* initialize a counter to display   */
                                 /* the line number

do until lines(filename) = 0     /* repeat this loop until no lines   */
                                 /* remain in the selected file...    */

   say lineno linein(filename)   /* display the line number, and then */
                                 /* read and display a line of text,  */
                                 /* advancing the read position       */
                                 /* with each pass though the loop    */

   lineno = lineno + 1           /* increment the line-number counter */

   end
exit                             /* end the program                   */
```

Figure 10-2. SHOLIN1.CMD

Resetting the Read Position

Have the user select how many lines are displayed. If a number larger then the number of lines in the file is typed, the program cycles back to the beginning. To do this, use LINEIN() with its second and third arguments. For example:

```
linein(stream,position,count)
```

where:

`stream` is the name of the stream from which the line is read.

`position` is the new position for the read position. The options are:

- no argument (the default)—to leave the read position where it is.
- 1—to set the read position to the beginning of the stream (line 1).

`count` is whether or not to actually read a line. The options are:

- 1 (the default)—to read one line and advance the read position.
- 0—to read no lines and not advance the read position.

So far, LINES() has been used with the default values for the second and third arguments to simply read the next line. By setting 1 for the `position` and 0 for the count, LINES() can be used to reset the read position to the beginning of the stream without reading a line (or advancing the read position). For example:

```
linein(filename,1,0)
```

Since you are not interested in the return value (which would be empty anyway), you can call LINEIN() as a subroutine:

```
call linein filename,1,0
```

Figure 10-3 shows an example calling LINEIN() as a subroutine to reset the read position.

```
/* Displays a given number of lines in a text file   */
/* If the given number exceeds the number of lines   */
/* in the file then the read position is reset back   */
/* to the beginning of the file                       */

say "Type a file name"
pull filename

say "Type number of lines to display"
pull howmany

lineno = 1
do howmany
   say lineno linein(filename)
   lineno = lineno + 1

   if lines(filename) = 0 then     /* if the end of file is reached */
     do
        call linein filename,1,0   /* reset the read position       */
        say ">>> End of File <<<"  /* display end-of-file marker    */
        lineno = 1                 /* reset line counter            */
        end
   end
exit
```

Figure 10-3. SHOLIN2.CMD

Printing a Text File

Using REXX to send data to a printer is very similar to writing to a file. There are some things you need to know:

- The name of the *device* that your computer is connected. Use this device name in place of a file name. In the previous example, the printer device is named PRN.
- The *control characters* and *escape sequences* that your printer uses for various formatting operations. Generally, these are listed in the manual for the printer. The example uses the standard character for a form feed, which directs the printer to *start a new page*.

Sending Special Characters

Printers often use certain characters as controls, such as ESC (escape) and FF (form feed) that you cannot type from the keyboard. To use these characters, you need to look up the ASCII number of the character. Then, you can use the built-in REXX function D2C() (decimal-to-character) to translate the ASCII code number into the character you need.

For example, the character that tells the printer to start a new page is ASCII number 12. To translate that code number 12 into a character for the printer use, you would write, `d2c(12)`.

For more information about functions such as D2C(), see "Using Functions to Convert Data" on page 4-19.

CHAROUT() Function

Generally, the CHAROUT() function is most useful for issuing printer controls.

```
CHAROUT(stream,string)
```

where:

`stream` is the name of the stream to which the character is written. Figure 10-4 on page 10-9 shows an example where `stream` is the name of the device connected to a printer.

`string` is a string—the characters to be written.

Like the LINEOUT function, CHAROUT() returns `0` if all the characters in `string` are successfully written to `stream`. Unlike LINEOUT(), if for any reason CHAROUT() cannot write to the named `stream`, it returns the number of characters that remain unwritten.

CHAROUT() is also similar to LINEOUT() in that it is more convenient to call it as a subroutine. For example, to tell the printer to start a new page, you could use `call charout "PRN",d2c(12)`.

A Printout Program

Figure 10-4 shows an example of a program for printing a file.

```
/* Prints a text file */
say 'Type file name: '          /* prompt for a file name            */
pull filename                    /* ...and get it from the user       */

printer = 'PRN:'                 /* save name of printer device       */
newpage = d2c(12)                /* save page-eject character         */

/* Repeat this loop until no lines remain in the file */
/* and keep track of the line count with COUNT       */

do count = 1 until lines(filename) = 0

/* Read a line from the file ...  */

   output =  linein(filename)

/* ... and send it to the printer */

   call lineout printer, output

   if result <> 0 then                              /* if there is a  */
      do                                            /* write error,   */
         say 'Error: unable to write to printer'    /* display it, and*/
         leave                                      /* exit the loop  */
         end

   if count // 50 = 0 then                /* if the line count is a */
      call charout printer, newpage       /* multiple of 50, then   */
                                          /* start a new page by    */
                                          /* sending the form feed  */

   end                                 /* go back to the start of loop    */
                                       /* until no lines remain           */

call lineout filename                  /* close the file                  */
exit                                   /* end the program normally        */
```

Figure 10-4. PRINTIT.CMD

To Summarize

Here is a review the input and output functions that have been used so far.

Use this function	To do this
`LINEOUT(stream,linedata)`	To open `stream` and append `linedata` (write it to the end of `stream`). Returns `1` if successful; `0` if otherwise.
`LINEOUT(stream)`	To close `stream` when writing is completed. Returns `1` if successful; `0` if otherwise. (REXX automatically closes any open streams at the end of a program.)
`LINEIN(stream)` or `LINEIN(stream,,1)`	To open `stream`, read the first line and advance the read position to the second line. If `stream` is already open, then LINEIN() reads the current line and advances the read position one line ahead.
`LINEIN(stream,1,0)`	To put the write position at the beginning of the `stream` *without* reading a line or advancing the read position.

Although not discussed in this chapter, you can also:

Use this function	To do this
`LINEIN(stream,1,1)`	To put the write position at the beginning of the `stream` *and* specifically read the first line (advancing the read position to the second line). The action is the same whether or not `stream` is already open.
`LINEIN(stream,,0)`	To open `stream` without reading the first line or advancing the read position. No action is taken if `stream` is already open.

Figure 10-5 is a table showing the REXX functions that read from and write to a data stream.

	Read	Write	Check for
Characters	CHARIN()	CHAROUT()	CHARS()
Lines	LINEIN()	LINEOUT()	LINES()

Figure 10-5. Read and Write Functions

STREAM() Function

You can use the STREAM() function to get the following information about a stream:

- To determine if a stream exists
- To determine if a stream is ready for input or output
- To get the size or last edit date of a file.

You can also use STREAM() for more complex input and output tasks, as when your program must read from or write to:

- Files other than text files
- Streams other than files and printers
- A specific position in a stream.

See "STREAM() Function" on page 10-18 for examples of STREAM().

Queues

A *queue* is a means of holding a list of lines in a specific order.

A queue in REXX combines the functions of a stack (in which the last item added is the first read) and a queue (in which the first item added is the first read). That is, it can be either *last in, first out (LIFO)* or *first in, first out (FIFO)*.

The REXX concept of a queue is different from that generally used. Throughout this book, the REXX definition a of queue is used.

One particular advantage of using a queue is that it lets your REXX programs share data with other programs, whether those programs are written in REXX or any other language.

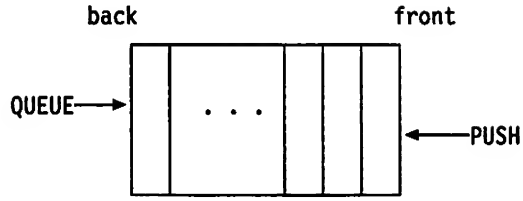
Data in a queue is held as a series of lines. Thus, data can only be put on or taken off a queue one line at a time. A line of data can be added to the back or to the front of a queue, but a line of data can only be taken off from the front.

The queue is external to REXX. Unlike variables, a queue remains even after your REXX program ends. Once created, it is retained for the duration of the current OS/2 session or until you delete it. This means that other programs can add or remove data from the queue whenever your REXX program allows.

Lists of Data

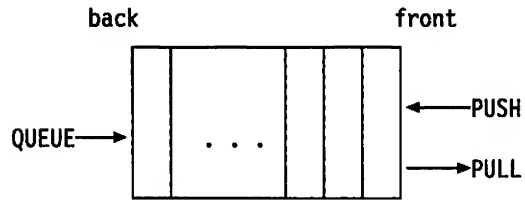
The difference between LIFO and FIFO *is* significant. Think of a queue as a list that you can add to from either end. You can add items on the back of the queue using the QUEUE instruction, or you can add items on the front of the queue, using the PUSH instruction.

You could picture it like this:



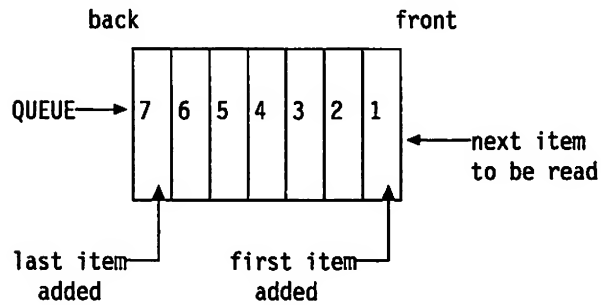
The concept of a queue having a *front* and a *back* is important, because you can only read items from the *front* of the queue. The command that reads data from a queue is the PULL instruction.

Now you could picture it like this:

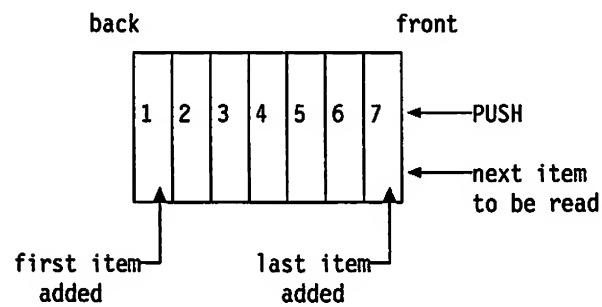


Putting Data on a Queue

The QUEUE instruction puts an item of data on the queue in FIFO order:



The PUSH instruction puts an item of data on the queue in LIFO order:



Reading the Queue

Up to now, the PULL instruction has been used to collect user input, such as data typed at the keyboard, to store it into variables.

What the PULL instruction actually parses is *the next available line of data from the queue*. It is only when the queue is empty (as it has been so far) that PULL parses input from the keyboard.

Try the program shown in Figure 10-6.

```
/* Using QUEUE and PULL */
QUEUE time()          /* add the current time to the queue  */
QUEUE date()          /* add the current date to the queue  */

PULL data1            /* get a string from the queue front  */
PULL data2            /* get a string from the queue front  */

say 'The first  element on the queue was:' data1
say 'The second element on the queue was:' data2

exit                  /* end the program normally           */
```

Figure 10-6. QUEUING.CMD

Change the program by substituting PUSH for QUEUE, see Figure 10-7.

```
/* Using PUSH and PULL */
push time()           /* add the current time to the queue  */
push date()           /* add the current date to the queue  */

pull Data1            /* get a string from the queue front  */
pull Data2            /* get a string from the queue front  */

say 'The first  element on the queue was:' Data1
say 'The second element on the queue was:' Data2

exit                  /* end the program normally           */
```

Figure 10-7. PUSHING.CMD

QUEUED Function

It would be useful to be able to determine how many items are on the queue at any given time. That is the purpose of the REXX function QUEUED, see Figure 10-8.

```
/* counting the queue */
push "manny"
push "moe"
push "mac"
count = queued()
say count "names in queue." /* displays '3 names in queue.' */
```

Figure 10-8. QCOUNT.CMD

You will find that QUEUED is particularly useful for controlling a DO loop that pulls data from the queue, see Figure 10-9.

```
/* Pull from the queue as long as items remain */
do until entry = ""
  say "Type name:"
  pull entry
  push entry
do for I while queued() > 0
   pull name.i
   say name.i
end
```

Figure 10-9. DOQUEUE.CMD

Summary

This completes "Basics" in this chapter. You have learned how to:

- Read and write disk files
- Send data to the printer
- Use the REXX data queue.

To make best use of the topics discussed in this chapter, see "Data Streams" in the *REXX Reference*.

"Advanced Topics" in this chapter discusses more about input and output.

To continue with "Basics," go to page 11-1.

Advanced Topics

In this chapter:

Advanced Topics

- More about data streams
- Default streams
- STREAM() function
- Accessing data within a stream
- More about queues.

More about Data Streams

REXX regards all information from any file or device as a continuous *stream* of single characters. Data read into a REXX program (whether from a disk file, the keyboard, a device, or another program) is processed equally as a character stream. The same is true for output data that a REXX program writes to a file or other device. All of these are streams.

Your program can work with the information in a stream as it is, 1 character at a time. Or, if the data is in line form (as in a text file), you can manipulate the information from the stream (or put information into it) line-by-line.

As shown in "Basics" in this chapter, your REXX programs can access and manipulate text files by using:

LINEIN()	to read a line
LINEOUT()	to write a line
LINES()	to check for the end of the stream
CHARIN()	to read 1 or more characters
CHAROUT()	to write 1 or more characters
CHARS()	to count the characters remaining in the stream.

Default Streams

Each of the I/O functions listed here has as its first argument the name of a stream that is read or written to. Each of these functions also has a *default stream* that is used if you omit the name of a specific stream.

STDIN	The default input stream (for LINEIN() and CHARIN() functions). STDIN is any data that is typed at the keyboard.
STDOUT	The default output stream (for LINEOUT() and CHAROUT() functions). Unless you have redirected it, STDOUT is the display and is the same stream that the SAY instruction uses.

This means that you can use the LINEIN() function to pause processing and read a line typed at the keyboard, as you can with the PARSE PULL instruction. But note these differences:

- Unlike PARSE PULL, the LINEIN() function reads *only* keyboard entries, regardless of whether there are outstanding items on the default data queue.
- LINEIN() does not prompt the user with a question mark.

To understand how this works, use the first file-reading program, SHOLIN1.CMD (see Figure 10-2 on page 10-6) and add an instruction as shown in Figure 10-10.

```
/* Displays a file one line at a time          */
/* as the user presses the Enter key; program  */
/* ends when the end-of-file is reached        */
/* OR if user types any character.             */

say "Type a file name"
pull filename
lineno= 1

do until lines(filename) = 0
   say lineno linein(filename)
   if linein() \= "" then leave   /* waits for user to press the Enter*/
                                  /* key; if anything else is typed */
                                  /* (if LINEIN() does not return  */
                                  /* an empty string), then the   */
                                  /* loop (and the program) ends   */
   lineno = lineno + 1
   end
exit
```

Figure 10-10. SHOLIN3.CMD

Or, you could modify the cycling version of this program, SHOLIN2.CMD (see Figure 10-3 on page 10-7) to let the user choose the number of lines to display. To do this, put the display routine inside a DO FOREVER loop, as shown in Figure 10-11.

```
/* Displays a file one line at a time   */
/* as the user presses the Enter key, or*/
/* displays a given number of lines.    */
/* Cycles back to the beginning of the  */
/* file when the end-of-file is reached */
/* Program ends only when user types    */
/* any non-numeric character.           */

say "Type a file name"
pull filename

say "Type a number of lines to display"
say "or press the Enter key alone to advance one line"
say "Type any other character to end."
lineno = 1

do forever
   howmany = linein()                  /* pause for user entry and store */
                                       /* it in the variable HOWMANY     */

   if howmany = "" then howmany = 1    /* pressing the Enter key alone   */
                                       /* is the same as typing '1'      */

   if \datatype(howmany,n) then leave/* typing any non-numeric          */
                                       /* character ends the program     */

   do howmany
      say lineno linein(filename)
      lineno = lineno + 1

      if lines(filename) = 0 then
        do
           call linein filename,1,0
           say ">>> End of File <<<"
           lineno = 1
           end
      end

   end
exit
```

Figure 10-11. SHOLIN4.CMD

Parsing Default Input

You can parse the input for individual words, either by using the instruction:

```
PARSE VALUE linein() WITH var1 var2 ...
```

or with the shorter form built into the PARSE instruction:

```
PARSE LINEIN var1 var2 ...
```

For more information about the PARSE instruction and its options, see Chapter 8, "Parsing." Also, see "Parsing" in the *REXX Reference*.

STREAM() Function

For more intricate and specialized input and output tasks, REXX provides another function called STREAM(). For example:

```
STREAM(name,operation,command)
```

where:

`name` is the stream you want to work on

`operation` is one of these:

- `C` for a command or action to be taken
- `S` for the state of the stream
- `D` for a more detailed description.

`command` is an action to perform. This argument must be used when and only when you specify `C` as the `operation`.

The syntax may look a bit complicated at first, because STREAM() has a wide variety of applications such as:

- The `C` (command) operation lets your program select and gain access to a named stream.
- The operations `S` and `D` (state and description) report the current status of a stream; that is, whether:
  - the stream is READY or NOTREADY for input/output
  - it is UNKNOWN (not yet identified)
  - an input or output ERROR has occurred.

For the full syntax of STREAM() and the other REXX input and output functions, see the *REXX Reference*.

Getting Information about a Stream

To determine if a particular stream exists, use the stream command QUERY EXIST with the STREAM() function call. For example:

```
stream(name,C,'query exists')
```

Note that the stream command is enclosed in matching quotes.

If the stream `name` exists, then this function call returns the full-path specification of the stream. For example:

```
C:\WORK\MYFILE.TXT
```

If the stream `name` does not exist, then the result is a null string.

Figure 10-12 shows an example of a program that reads a file.

```
/* For a program that reads a file       */
say "Type a file name (or press the Enter key alone to exit): "
pull fname
if fname = "" then exit

/* Check that the file exists:           */
/* if STREAM() returns a null string,    */
/* then report the stream not found      */
/* and exit....                          */

call stream fname, C, 'query exists'
if result = "" then
   do
     say "Can not find" fname"."
     say "Check for proper path specification."
     exit
   end

/* ...else store the full pathname       */
/* (in RESULT) to the variable FNAME,    */
/* in case the user has typed a          */
/* relative path (e.g., "..\docs\my.txt" */

else fname = result
say "Full path specification is" fname
```

Figure 10-12. QRYFILE1.CMD

You can also query for information about the size of a stream and the date and time of the last edit, see Figure 10-13.

```
/* How big and when last changed? */
say "Type a file name (or press the Enter key alone to exit): "
pull fname
:
bytes = stream(fname,c,'query size')
ledit = stream(fname,c,'query datetime')
say fname "is" bytes "bytes."
say "Last edit of" fname "was" ledit"."
```

Figure 10-13. QRYFILE2.CMD

Opening and Closing Streams

The functions LINEOUT(), LINEIN(), CHARIN() and CHAROUT() do much of their own *housekeeping*. They automatically open the streams they work on and leave REXX to close the stream at the end of a program.

However, there are cases where it is necessary (or at least more prudent) to explicitly open and close a stream, such as in a program that reads from more than one device or one that writes to the middle of a file.

This is done with the STREAM() function:

```
stream(name,c,"open")
```

This default form opens a stream name for both reading and writing text. To open a stream for:

- Writing only, add the word `write`. For example:

  ```
  stream(name,c,"open write")
  stream(name,c,"open")
  ```

- Reading only, add the word read. For example:

  ```
  stream(name,c,"open read")
  ```

When you open a stream in this way, STREAM() returns the string `READY:` if the stream has been successfully opened. An error message is returned if for any reason it was unable to open the stream.

To explicitly close a stream, use:

```
stream(name,c,"close")
```

In this form, STREAM() returns a null string ("") if the operation is successful, the string `ERROR` if the operation fails.

Accessing Data within a Stream

REXX regards all external data as streams of information. Nonetheless, these streams can take different forms. A disk file, for example, differs from the output to a printer in that it has a static, physical form. A disk file is one example of a *persistent* stream. This means a disk file can not only be read from its beginning or written to its end, it can be read from and written to any place between.

As a program reads a file, REXX keeps a place marker, called the *read position*, that points to the next character (or line) to be read.

The same is true when writing. REXX maintains a *write position*, separate from the read position, that marks the next place it is to write.

When both reading and writing a file, the read position and the write position are the same.

If you do not specify a position for these markers, REXX advances them, by default, the number of characters read or written.

You can specify another position for the read or write positions by giving additional arguments to the stream functions, LINEIN(), LINEOUT(), CHARIN(), CHAROUT(), or by using the `position` option of the STREAM() function. For more information, refer to "Functions" in the *REXX Reference*.

More about Queues

In certain applications, your REXX programs can organize information for use by other programs by putting it in an external data queue. A *queue* is an ordered list that can be read or written at either end, top or bottom, so that each item of the queue constitutes a line of data.

There are two kinds of queues in REXX:

- One default queue, SESSION, is automatically provided for each active OS/2 session. SESSION is created by REXX the first time a REXX program issues a PUSH or QUEUE instruction to a line of data. Any program, REXX or otherwise, in a given session can access the SESSION queue, but only the SESSION queue defined for its own session can be accessed.
- A REXX program can also create private queues for itself. A private queue must be accessed by a unique name. You can name the queue or leave the naming to REXX.

Private data queues are created and manipulated by the RXQUEUE function, described in "Applications Programming Interfaces" in the *REXX Reference*.

For more information about file and device input and output, refer to "Functions" and "Data Streams" in the *REXX Reference*.

Examples

```
/* SDIR.CMD - Program to print a sorted directory.          */
/*   Program will read in the current directory, and       */
/*     then sort it using a quick-sort routine.            */
list. = 0
'dir ¦ rxqueue /fifo'
j=0
do queued()
  j=j+1
  parse pull list.j
  end
call quicksort 4, j-1
do i = 1 to j
   say list.i
end
exit 0
/* The quick-sort procedure */
Quicksort:
PROCEDURE EXPOSE list.
ARG bot, top
center = Qsort(bot, top)
IF center - 1 > bot THEN CALL Quicksort bot, center - 1
IF center + 1 < top THEN CALL Quicksort center + 1, top
RETURN
```

Figure 10-14 (Part 1 of 2). SDIR.CMD

```
qsort: PROCEDURE EXPOSE list.
ARG b , t
choose = list.b
small = b
large = t + 1
DO WHILE (small + 1 < large)
   next = small + 1
   IF list.next <= choose THEN
      DO
      list.small = list.next
      small = small + 1
      list.next = choose
      END
   ELSE
      DO
      large = large - 1
      temp = list.large
      list.large = list.next
      list.next = temp
      END
   END
RETURN small
```

Figure 10-14 (Part 2 of 2). SDIR.CMD

```
/* Program printing exec - program will determine which records  */
/*   in a file exceed the specified length.                      */

say 'Please type the name of the file to examine {filename.ext}: '
parse pull file
say 'Please type the length to scan for: '
parse pull col
"type" file "¦ rxqueue /fifo"
if rc <> 0 then exit rc
lines = queued()
do currline = 1 to lines
   parse pull line
   if length(line) > col then
     say 'Line #' currline '  length =' length(line)
   end
exit 0
```

Figure 10-15. LINLEN.CMD

Chapter 11. Program Style

The focus in this chapter is more on *method* than on individual features, which are better learned by practicing and experimenting.

Basics

In this chapter:

Basics

- Consider the data
- Define the tasks
- Create modules
- Planning the program
- Putting it all together
- Testing and debugging.

As you learn more about the syntax of REXX, you can get better at deciding which computing tasks are appropriate to a program. Translating an idea for a program into actual code is less a matter of expert programming than of good planning. If you plan your program thoroughly, the coding will be that much easier.

Consider the Data

When you are faced with the task of writing a program, the first thing to consider is the data you are required to process.

1. Make a list of the input data. What are the items and the possible values of each?
2. If the input data items have a type of structure or pattern, draw a diagram to illustrate it.
3. Do the same for the output data. What data and in what form does the user expect as output?
4. Study your two diagrams to see if they fit together. If they do, you are well on the way to designing your program.
5. Write a specification of input for the user. This may be a written specification, a HELP file, or both.

Test Yourself

You are required to write an interactive program that invites the user to play *heads or tails*. The game can be played as long as the user likes. To end the game, the user types `Quit` in answer to the question `Heads or Tails?`. The program is arranged so that the computer *always* wins.

Think about how you would write this program.

The computer starts with:

```
Let us play a game!  Type "Heads", "Tails", or "Quit"
and press the Enter key.
```

This means that there are *four* possible inputs:

- Heads
- Tails
- Quit
- None of these three.

And so the corresponding outputs should be:

- Sorry. It was TAILS. Hard luck!
- Sorry. It was HEADS. Hard luck!
- (no output)
- That is not a valid answer. Try again!

This sequence must be repeated indefinitely, ending with the return to the OS/2 program.

Now that you understand the specification, the input data, and the output data, you are ready to write the program.

Write the program. If you are careful, it should run the first time.

Answer: Figure 11-1 is an example of a possible solution.

```
/* Tossing a coin.  The machine is lucky, not the user */

do forever
   say "Let us play a game!  Type 'Heads', 'Tails'",
       "or 'Quit' and press the Enter key."
   pull answer

   select
      when answer = "HEADS"
         then say "Sorry!  It was TAILS.  Hard luck!"
      when answer = "TAILS"
         then say "Sorry!  It was HEADS.  Hard luck!"
      when answer = "QUIT"
         then exit
      otherwise
         say "That is not a valid answer.  Try again!"
   end
   say
end
```

Figure 11-1. CON.CMD

Define the Tasks

You are going to knit a warm, woolen, pullover sweater to wear when you go sailing. You may:

1. Knit the front
2. Knit the back
3. Knit the left arm
4. Knit the right arm
5. Sew the pieces together.

Each of these jobs is simpler to describe than the the job of knitting. In computer jargon, separating a job into simpler jobs is called *stepwise refinement.*

Look at the specification again. You may need to put on the pullover in the dark quickly, without worrying about the front or back. Therefore, the front should be the same as the back, and the two sleeves should also be the same. Figure 11-2 shows a way that this could be coded.

```
do 2
   CALL Knit_body_panel
end

do 2
   CALL Knit_sleeve
end

CALL sew_pieces_together
```

Figure 11-2. PULLOVER.CMD

Reconsider the Data

When you are refining your program, your objective is to make each piece simpler. This means simpler:

- Input data for each segment or routine
- Output data for each segment or routine
- Processing
- Code.

In the preceding example, if your pieces really are simpler, they probably have simpler names, too. For example:

- Knit cuff.

  rather than

- Make ribbing for cuffs and waistband.

Create Modules

Using the stepwise refinement method discussed previously, you start with a specification (which may be incomplete). Then, you separate the proposed program into routines, so that each routine is easier to code than the entire program. You repeat the process for each of these routines until you reach routines that you are sure you can code correctly the first time.

While you are doing this, keep asking yourself two questions:

- What data does this routine handle?
- Is the specification complete?

Still thinking about *method*, which is as important as *language*, look at a simple arcade-type game program called CATMOUSE.CMD, as shown in Figure 11-3. You may want to take a moment to type it in and play it.

```
/* The user says where the mouse is to go.  But where   */
/* will the cat jump?                                   */

say "This is the mouse ---------->   @"
say "These are the cat's paws --->  ( )"
say "This is the mousehole ------>   0"
say "This is a wall ------------>   ¦"
say
say "You are the mouse.  You win if you reach",
    "the mousehole.  You cannot go past"
say "the cat.  Wait for him to jump over you.",
    "If you bump into him you are caught!"
say
say "The cat always jumps towards you, but he's not",
    "very good at judging distances."
say "If either player hits the wall he misses a turn"
say
say "Type a number between 0 and 2 to say how far to",
    "the right you want to run."
say "Be careful, if you type a number greater than 2 then",
    "the mouse will freeze and the cat will move!"
say
```

Figure 11-3 (Part 1 of 3). CATMOUSE.CMD

```
/*---------------------------------------------------*/
/* Parameters that can be changed to make a different  */
/* game                                               */
/*---------------------------------------------------*/
len = 14              /* length of corridor           */
hole = 14             /* position of hole             */
spring = 5            /* maximum distance cat can jump */
mouse = 1             /* mouse starts on left         */
cat = len             /* cat starts on right          */
/*---------------------------------------------------*/
/* Main program                                       */
/*---------------------------------------------------*/
do forever
   call display
   /*------------------------------------------------*/
   /* Mouse's turn                                   */
   /*------------------------------------------------*/
   pull move
   if datatype(move,whole) & move >= 0 & move <= 2
   then select
      when mouse + move > len then nop     /* hits wall */
      when cat > mouse,
         & mouse + move >= cat             /* hits cat  */
                                        /* continued ... */
      then mouse = cat
      otherwise                         /* moves     */
      mouse = mouse + move
   end
   if mouse = hole then leave          /* reaches hole */
   if mouse = cat then leave           /* hits cat     */
   /*------------------------------------------------*/
   /* Cat's turn                                     */
   /*------------------------------------------------*/
   jump = random(1,spring)
   if cat > mouse then do     /* cat tries to jump left */
      if cat - jump < 1 then nop          /* hits wall  */
      else cat = cat - jump
   end
   else do                  /* cat tries to jump right */
      if cat + jump > len then nop        /* hits wall  */
      else cat = cat + jump
   end
   if cat = mouse then leave
end
```

Figure 11-3 (Part 2 of 3). CATMOUSE.CMD

```
/*---------------------------------------------------------*/
/* Conclusion                                              */
/*---------------------------------------------------------*/
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit
/*---------------------------------------------------------*/
/* Subroutine to display the state of play                 */
/*                                                         */
/* Input: CAT and MOUSE                                    */
/*                                                         */
/* Design note:  each position in the corridor occupies */
/* three character positions on the screen.                */
/*---------------------------------------------------------*/
display:
corridor = copies(" ",3*len)                /* corridor  */
corridor = overlay("O",corridor,3*hole-1)  /* hole      */

if mouse \= len                          /* mouse in hole? */
then corridor = overlay("@",corridor,3*mouse-1)/* mouse */

corridor = overlay("(",corridor,3*cat-2)       /* cat   */
corridor = overlay(")",corridor,3*cat)
say "   ¦"corridor"¦"
return
```

Figure 11-3 (Part 3 of 3). CATMOUSE.CMD

Planning the Program

The program is about a cat and a mouse and their positions in a corridor.

1. The program begins with some initial settings, such as the length of the corridor and the positions of the *cat* and the *mouse*.
2. The player types the *moves* of the mouse. The cat's jumps are generated using a random number. The resulting positions are calculated at some point.
3. At some other point, these positions are displayed on the screen.

Obviously, the whole program is too complicated to think about all at once. The first step is to separate it into *tasks* such as:

- Setup—establish the initial positions.
- Main program—accept and calculate the result of each move.
- Display subroutine—display the new positions.

Now look at the main program. The user (who plays the mouse) will want to see where everybody is before making a move. The cat will not. The next step is to separate the main program into:

```
Do forever
   call Display
   Mouse's move
   Cat's move
End
Conclusion
```

Designing Loops

The method for designing loops is to ask two questions:

- Will it always terminate?
- Whenever it terminates, will the data meet the conditions required?

The loop terminates (and the game ends) when:

- The mouse runs to the hole.
- The mouse runs into the cat.
- The cat catches the mouse.

Conclusion

At the end of the program, the user must be told what happened by issuing the following:

```
call display
say who won
```

Putting It All Together

Figure 11-4 shows how to put the modules together:

```
/*---------------------------------------------------*/
/* Main program                                      */
/*---------------------------------------------------*/
do forever
   call display
   /*------------------------------------------------*/
   /* Mouse's turn                                   */
   /*------------------------------------------------*/
    ...

   if mouse = hole then leave          /* reaches hole */
   if mouse = cat then leave           /* hits cat     */
   /*------------------------------------------------*/
   /* Cat's turn                                     */
   /*------------------------------------------------*/
    ...

   if cat = mouse then leave
end

/*---------------------------------------------------*/
/* Conclusion                                        */
/*---------------------------------------------------*/
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit

/*---------------------------------------------------*/
/* Subroutine to display the state of play           */
/* Input: CAT and MOUSE                              */
/*---------------------------------------------------*/
display:
 ...
```

Figure 11-4. CATMOUSE2.CMD

Testing and Debugging

If you cannot understand why your program is giving wrong results, you can:

- Modify your program so that it tells you what it is doing
- Put extra instructions into your program, such as:

```
:
say "Checkpoint A.  x =" x
:
say "End of first routine"
:
```

- Use some of the REXX interactive trace facilities.

You will gradually learn which of these techniques is best for you.

Using TRACE

The TRACE instruction is a facility that you can use to perform the following tasks.

- To find out where your program is going, use TRACE L (labels). Figure 11-5 shows an example and the trace it displays on the screen.

```
/* Example: two iterations of wheel, six iterations */
/* of cog.  On the first three iterations,  "x < 2" */
/* is true. On the next three, it is false.        */
trace L
do x = 1 to 2
wheel:
   do 3
cog:
      if x < 2 then do
true:
      end
      else do
false:
      end
   end
end
done:
```

Figure 11-5. ROTATE.CMD

The following trace is displayed on the screen.

```
[C:\]rotate
     6 *-*  wheel:
     8 *-*   cog:
    10 *-*     true:
     8 *-*   cog:
    10 *-*     true:
     8 *-*   cog:
    10 *-*     true:
     6 *-*  wheel:
     8 *-*   cog:
    13 *-*     false:
     8 *-*   cog:
    13 *-*     false:
     8 *-*   cog:
    13 *-*     false:
    17 *-* done:
[C:\]
```

- To see how the interpreter is computing expressions, use TRACE I (intermediates).
- To find out whether you are passing the right data to a command or subroutine, use TRACE R (results).
- To make sure that you get to see nonzero return codes from commands, use TRACE E (errors).

TRACE Symbols

The trace symbols mean:

- Identifies the source of a single clause, that is, the data actually in the program.

+++ Identifies a trace message. This may be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error when in interactive debug, or the traceback clauses after a syntax error in the program.

>>> Identifies the result of an expression (for TRACE R), the value assigned to a variable during parsing, or the value returned from a subroutine call.

>.> Identifies the value assigned to a placeholder during parsing (see "Using a Placeholder" on page 8-7).

If you are using TRACE I (intermediates), these symbols are also used:

>C> The data traced is the name of a compound variable, traced after substitution and before use, provided that the name had the value of a variable substituted into it.

>F> The data traced is the result of a function call.

>L> The data traced is a literal (string, uninitialized variable, or constant symbol).

>O> The data traced is the result of an operation on two terms.

>P> The data traced is the result of a prefix operation.

>V> The data traced is the contents of a variable.

Interactive Debugging

By putting a question mark in front of the TRACE option, `TRACE ?R`, you can turn on the REXX *interactive debugging* tool. That means the program pauses after it processes most instructions (exceptions include SIGNAL, CALL, and reiterations of DO loops). You can examine each clause, one at a time, and advance processing by pressing the Enter key.

For more information about interactive debug, refer to "Debug Aids" in the *REXX Reference*.

Summary

This completes "Basics" in this chapter. You have learned how to:

- Define tasks and create program modules
- Plan and develop a program
- Test and debug a program.

"Advanced Topics" in this chapter discusses refining programs to make them easier to read.

Advanced Topics

In this chapter:

Advanced Topics

- Making programs easy to read.

Making Programs Easy to Read

The only sure way to determine if a program is correct is to read it. Therefore, programs must be easy-to-read. Easy to read means different things to different programmers. Here are examples of different styles. You can choose the style you prefer.

A very good way to check your program is to ask someone to read it. Be sure to choose a coding style that they find easy to read.

Most people would find the program fragment shown in Figure 11-6 difficult to read.

```
/*************************************************************/
/* SAMPLE #1:  A portion of CATMOUSE.CMD (Page 11-4),        */
/* not divided into segments and written with no indentation  */
/* and no comments.  This style is not recommended.          */
/*************************************************************/
do forever
call display
pull move
if datatype(move,whole) & move >= 0 & move <=2
then select
when mouse+move > len then nop
when cat > mouse,
& mouse+move >= cat,
then mouse = cat
otherwise
mouse = mouse + move
end
if mouse = hole then leave
if mouse = cat  then leave
jump = random(1,spring)
if cat > mouse then do
if cat-jump < 1 then nop
else cat = cat-jump
end
else do
if cat+jump > len then nop
else cat = cat+jump
end
if cat = mouse then leave
end
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit
```

Figure 11-6. CATMOUSE3.CMD

Figure 11-7 shows an example that is easier to read. It is separated into segments, each with its own heading. The comments on the right are sometimes called *remarks*. They can help the reader get a general idea of what the program is doing.

```
/*********************************************************/
/* SAMPLE #2:  A portion of CATMOUSE.CMD (Page 11-4),    */
/* divided into segments and written with 'some'         */
/* indentation and 'some' comments.                      */
/*********************************************************/

/*********************************************************/
/* Main program                                          */
/*********************************************************/
do forever
   call display
   /*****************************************************/
   /* Mouse's turn                                      */
   /*****************************************************/
   pull move
   if datatype(move,whole) & move >= 0 & move <=2
   then select
      when mouse+move > len then nop       /* hits wall */
      when cat > mouse,
         & mouse + move >= cat,            /* hits cat  */
      then mouse = cat
      otherwise                            /* moves     */
      mouse = mouse + move
  end
  if mouse = hole then leave             /* reaches hole */
  if mouse = cat then leave              /* hits cat     */
  /*****************************************************/
  /* Cat's turn                                        */
  /*****************************************************/
  jump = random(1,spring)
  if cat > mouse then do      /* cat tries to jump left */
     if cat - jump < 1 then nop          /* hits wall  */
     else cat = cat - jump
  end
  else do                   /* cat tries to jump right */
     if cat + jump > len then nop        /* hits wall  */
     else cat = cat + jump
  end
  if cat = mouse then leave
end
/*********************************************************/
/* Conclusion                                            */
/*********************************************************/
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit
```

Figure 11-7. CATMOUSE4.CMD

Figure 11-8 shows an example with additional features that are popular with some programmers. Keywords written in uppercase and a different indentation style highlight the structure of the code. The abundant comments recall the detail of the specification.

```
/*********************************************************/
/* SAMPLE #3:  A portion of CATMOUSE.CMD (Page 11-4),    */
/* divided into segments and written with 'more'         */
/* indentation and 'more' comments.                      */
/* Note commands in uppercase (to highlight logic)       */
/*********************************************************/

/*********************************************************/
/* Main program                                          */
/*********************************************************/
DO FOREVER
  CALL display
  /*********************************/
  /* Mouse's turn                  */
  /*********************************/
  PULL move
  IF datatype(move,whole) & move >= 0 & move <=2
    THEN SELECT
           WHEN mouse+move > len     /* mouse hits wall */
             THEN nop                /* and loses turn  */
           WHEN cat > mouse,
                & mouse+move >= cat, /* mouse hits cat  */
             THEN mouse = cat        /* and loses game  */
           OTHERWISE mouse = mouse + move /* mouse ...  */
         END                       /* moves to new location */
  IF mouse = hole THEN LEAVE   /* mouse is home safely  */
  IF mouse = cat  THEN LEAVE   /* mouse hits cat (ouch) */
  /*********************************/
  /* Cat's turn                    */
  /*********************************/
  jump = RANDOM(1,spring)      /* determine cat's move  */
  IF cat > mouse               /* cat must jump left    */
    THEN DO
           IF cat-jump < 1          /* cat hits wall    */
             THEN nop               /* misses turn      */
             ELSE cat = cat-jump    /* cat jumps left   */
         END
    ELSE DO                     /* cat must jump right   */
           IF cat+jump > len        /* cat hits wall    */
             THEN nop               /* misses turn      */
             ELSE cat = cat+jump    /* cat jumps right  */
         END
  IF cat = mouse THEN LEAVE    /* cat catches mouse     */
END

                               /*continued*/
```

Figure 11-8 (Part 1 of 2). CATMOUSE5.CMD

```
/*********************************************************/
/* Conclusion                                            */
/*********************************************************/
CALL display                          /* on final display */
  IF cat = mouse                      /* who won?         */
    THEN say "Cat wins"               /* ... the cat      */
    ELSE say "Mouse wins"             /* ... the mouse    */
EXIT
```

Figure 11-8 (Part 2 of 2). CATMOUSE5.CMD

Index

E

F

G

O

P

Q

R

S

T

U

V

W

Special Characters

64F3058

Order No. 01F0272
S01F-0272

9001F02720001